TECHNISCHE
UNIVERSITÄT
DARMSTADT

Composing Non-Functional Concerns in Web Services

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von
Diplom-Informatiker

Benjamin Schmeling
geboren in Wiesbaden, Deutschland

Referent: Prof. Dr.-Ing. Mira Mezini, Technische Universität Darmstadt

Korreferent: Prof. Dr.-Ing. Stefan Tai, Karlsruher Institut für Technologie

Tag der Einreichung: 5. 3. 2013
Tag der mündlichen Prüfung: 5. 6. 2013

Erscheinungsjahr 2013

Darmstadt D17

Affirmation / Ehrenwörtliche Erklärung

I hereby declare that I have written the following thesis without the inadmissible assistance of third parties and using only the indicated sources and aids. All instances in which outside sources were used have been marked accordingly. This work has not been presented to any test authority in its current or in a similar form.

Hiermit versichere ich, die vorliegende Doktorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, March 2013 Benjamin Schmeling

Acknowledgments

My sincere thanks go to Prof. Dr. Mira Mezini and Prof. Dr. Tai for their support and guidance during my Ph.D. work as well as to my colleague Anis Charfi for his advice, guidance and help throughout the rendering of this Ph.D. thesis. He contributed much of his time in fruitful discussion with me about new ideas and their realization.

I am also grateful for the excellent support from SAP Research and my colleague Steffen Heinzl for the many valuable discussions on my Ph.D. thesis, and in this regard I would also like to thank Mohamed Aly, Heiko Witteborg, Francesco Novelli and all the other colleagues.

My students Rainer Thome, Marko Martin and Ajay Sagar contributed to the work described in this thesis by implementing parts of the modeling tool and code generator in the context of their theses.

A well-deserved word of appreciation is also in order for Evelyn Nichols for her efforts to improve my English.

Finally, I extend my profound gratitude to my family for their unwavering support throughout my studies, which ultimately helped to make their successful conclusion and this thesis a reality.

I authored this paper while working as a research associate at SAP Research, during which time I contributed to projects PREMIUM|Services (01IA08003A) and InDiNet (01IC10S04A) which were partially funded by the German Federal Ministry of Education (BMBF). Hence, I thank the BMBF for its financial support.

Abstract

In software development, two types of concerns must generally be addressed: functional and non-functional ones. Functional concerns relate to the main or core functionality of a software, whereas non-functional concerns represent quality characteristics of the same. It is widely accepted that the two types of concerns should be strictly separated from one another to increase maintainability, understandability and reusability of a software. In component-based software systems, this strict separation of concerns results in the implementation of the respective concerns as functional as well as non-functional components.

From the current perspective, web services are the prevalent technology for implementing such component-based software systems. They adhere to open standards and describe a well-defined interface which is strictly separated from the implementation of the service. This allows the realization of systems consisting of loosely coupled and platform-independent services. Non-functional concerns could also be implemented as web services; however, a well-defined mechanism is required to integrate these non-functional components with functional ones. This mechanism should not interfere with the aforementioned platform independence and loose coupling, and it should not require changes in the implementation of the functional component. For reuse purposes, it should be possible to integrate one non-functional component with several functional components. Furthermore it should be possible to integrate multiple non-functional components with a single functional component. The latter, however, requires a well-defined ordering of the consumption of the non-functional components. The justification for this is that, generally,different orderings of non-functional concerns would cause different behavior. Furthermore, it is not possible to define generally valid (default) orderings, because the ordering may be specific to a particular functional component. Thus, a mechanism for specifying these orderings explicitly is required. Furthermore, a component must be available which is able to enforce this specification at runtime.

This dissertation analyzes the applicability of state-of-the-art approaches for the composition of non-functional concerns in web services in terms of the concrete requirements to be met. In this analysis it turns out that there is no appropriate approach which sufficiently supports most of the requirements. Hence, the dissertation presents NFComp, a novel, model-driven approach for composing non-functional concerns in web services. This approach takes different dimensions into account. Firstly, it aims at both specification as well as enforcement of concern compositions. Secondly, it covers different views on web services in which only particular parts of the service are available. Thirdly, it provides an abstract framework which could be applied to all component-based approaches and a concrete, instantiable one which can be applied to web services in particular.

In this approach, a modeler can specify concern composition in terms of non-functional actions each representing distinct and fine-grained non-functional behavior. Those actions, their logical

composition and mapping to functional components, can be modeled in a graphical way. The model is mainly process-oriented and shows directly in which order and for which services actions must be executed. The approach is structured in six phases: requirements specification, action definition, action composition, service mapping, middleware mapping and code generation. In each phase, the model is processed by various participants in different roles and enriched by new information.

The resulting model can be validated at design time against a set of constraints imposed by different types of interdependencies modeled in the action definition phase. In this regard, the problem of finding interdependencies which crosscut different non-functional domains has been addressed. A mechanism has been defined to systematically analyze data dependencies to infer control flow constraints. This mechanism helps to enrich the modeled set of interdependencies by discovering even cross-domain interdependencies and thus enables a more precise validation. In addition to the classical validation where constraint violations are directly shown in the model, a guided modeling procedure has been invented. This procedure supports the modeler by showing the next safe modeling steps always resulting in a valid model. Finally, a generator takes the model as input in order to produce code which enforces the modeled composition at runtime. The generator, however, does not produce the implementation of non-functional components which must be implemented manually. Instead, it provides a composition component which takes over the task of integrating and invoking these components in the specified order according to the model. The whole approach has been implemented for web services as a set of Java-based modeling tools and a code transformer which generates either the configuration for an Enterprise Service Bus or, alternatively, a set of aspects implementing the composition component. The aspect-based approach is not only applicable to web services but also generally to component-based software written in Java.

Zusammenfassung

In der Softwareentwicklung wird im Allgemeinen zwischen funktionalen und nicht-funktionalen Belangen (Concerns) unterschieden. Funktionale Belange beziehen sich auf die Haupt- oder Kernfunktionalität einer Software, wohingegen nicht-funktionale Belange qualitative Eigenschaften repräsentieren. Es wird allgemein akzeptiert, dass funktionale und nicht-funktionale Belange strikt voneinander getrennt werden sollten, um die Wartbarkeit, Verständlichkeit und Wiederverwendbarkeit einer Software zu erhöhen. In komponentenbasierten Softwaresystemen führt diese strikte Trennung sowohl zu funktionalen, als auch zu nicht-funktionalen Komponenten.

Web Services sind die, aus heutiger Sicht, gängigste Technologie, um solche komponentenbasierte Softwaresysteme zu implementieren. Web Services verwenden offene Standards und beschreiben wohldefinierte Schnittstellen, die strikt von der Implementierung getrennt sind. Dies ermöglicht es Systeme zu realisieren, die aus lose gekoppelten, plattformunabhängigen Diensten bestehen. Auch nicht-funktionale Belange könnten als solche Dienste realisiert werden, jedoch wird hierzu ein wohldefinierter Mechanismus benötigt, der es erlaubt die nicht-funktionalen mit den funktionalen Komponenten zu integrieren. Dieser Mechanismus sollte weder die Plattformunabhängigkeit noch die lose Kopplung negativ beeinflussen. Weiterhin sollte dieser keine Änderung der bestehenden Funktionalität einer funktionalen Komponente erfordern. Um Wiederverwendbarkeit zu gewährleisten, sollte man zudem ein und dieselbe nicht-funktionale Komponente mit mehreren funktionalen Komponenten und umgekehrt eine funktionale mit mehreren nicht-funktionalen Komponenten kombinieren können. Letzteres erfordert eine wohldefinierte Aufrufreihenfolge der jeweiligen nicht-funktionalen Komponenten. Dies ist in erster Linie damit zu begründen, dass im Allgemeinen unterschiedliche Reihenfolgen zu verschiedenen Ergebnissen führen. Weiterhin ist es nicht möglich eine allgemein gültige Reihenfolge zu definieren, da diese in der Regel spezifisch für eine bestimmte funktionale Komponente ist. Deshalb wird ein Mechanismus benötigt, der es erlaubt solche Reihenfolgen explizit zu spezifizieren. Dieser muss letztendlich durch eine Komponente ergänzt werden, die dafür verantwortlich ist, eine solche Spezifikation zur Laufzeit durchzusetzen.

In dieser Doktorarbeit wird die Anwendbarkeit von aktuellen Ansätzen für die Komposition von nicht-funktionalen Belangen in Web Services, mittels konkreter zu erfüllender Anforderungen, analysiert. Das Ergebnis dieser Analyse zeigt, dass es bisher keinen geeigneten Ansatz gibt, der die meisten der Anforderungen in ausreichendem Maße erfüllt. Deshalb wird in dieser Doktorarbeit NFComp, ein neuer, modellgetriebener Ansatz für die Komposition von nicht-funktionalen Belangen in Web Services, vorgestellt. Dabei werden vorwiegend drei unterschiedliche Ziele verfolgt: Erstens zielt NFComp sowohl auf die Spezifikation als auch die Durchsetzung dieser Belange ab. Zweitens deckt NFComp unterschiedliche Sichten auf Web Services ab, in denen nur bestimmte Teile des Services zur Verfügung stehen. Drittens wird sowohl eine abstraktes Rahmenwerk vorgestellt, welches auf alle komponentenbasierten Ansätze angewendet werden

kann, als auch ein konkreter, instanziierbarer Ansatz für Web Services im Speziellen.

In dem vorgestellten Ansatz kann ein Modellierer die Komposition von verschiedenen Belangen in Form von nicht-funktionalen Aktionen, die unterschiedliches, abgrenzbares, feingranulares nicht-funktionales Verhalten repräsentieren, spezifizieren. Diese Aktionen, ihre logische Komposition und ihre Assoziation mit funktionalen Komponenten kann dabei graphisch modelliert werden. Das resultierende Modell ist vorwiegend prozessorientiert und stellt direkt dar, in welcher Reihenfolge und für welche Web Services diese Aktionen ausgeführt werden müssen. Der Ansatz ist in sechs Phasen gegliedert: Anforderungsspezifikation, Definition von Aktionen, Aktionen Komposition, Assoziation von Aktionen mit Diensten, Abbildung von Aktionen auf Middlewaredienste und Codegenerierung. In jeder Phase, wird das Modell von unterschiedlichen Benutzern in verschiedenen Rollen bearbeitet und mit neuen Informationen angereichert.

Das resultierende Gesamtmodell kann zur Designzeit gegen eine Menge von Einschränkungen (Constraints), die durch verschiedene Arten von in der Aktionendefinitionsphase modellierte Abhängigkeiten (Interdependencies) eingeführt worden sind, validiert werden. In diesem Zusammenhang wurde auch das Problem, Abhängigkeiten zwischen Aktionen, die aus verschiedenen nicht-funktionalen Domänen stammen, zu bestimmen, gelöst. Es wurde ein Mechanismus definiert, der systematisch Datenabhängigkeiten analysiert, um Kontrollflusseinschränkungen daraus zu folgern. Dieser Mechanismus ermöglicht es, das Modell mit neuen Abhängigkeiten, auch solchen zwischen unterschiedlichen nicht-funktionalen Domänen, anzureichern, was eine präzisere Validierung gewährleistet. Zusätzlich zur klassischen Validierung, in der Verletzungen von Einschränkungen direkt im Modell hervorgehoben werden, wurde eine benutzergeführte Modellierungsprozedur konzipiert und umgesetzt. Diese Prozedur unterstützt den Modellierer, indem diese ihm die nächsten sicheren Modellierungsschritte anzeigt, welche immer in einem validen Modell resultieren. Letztendlich dient das Modell dann einem Generator als Eingabe, um daraus Code zu generieren, der das modellierte Kompositionsverhalten zur Laufzeit durchsetzt. Der Generator produziert dabei jedoch nicht die Implementierung der nicht-funktionalen Komponenten. Diese müssen manuell implementiert werden. Stattdessen wird eine Kompositionskomponente zur Verfügung gestellt, die die Aufgabe übernimmt, die nicht-funktionalen Komponente zu integrieren und sie in der Reihenfolge aufzurufen, die im Modell spezifiziert wurde. Der gesamte Ansatz wurde als eine Menge von Modellierungstools und Code Generatoren für Web Services implementiert. Der Code Generator kann, je nach Anforderung eine Konfiguration für einen Enterprise Service Bus oder aber als eine Menge von Aspekten, welche dann zusammen die Kompositionskomponente realisieren, erzeugen. Der aspekt-orientierte Ansatz ist nicht nur auf Web Services anwendbar, sondern auf in Java entwickelte, komponentenbasierte Software im Allgemeinen.

Contents

1 Introduction **1**

1.1 Thesis Outline 1

1.2 Contributions 3

1.3 Structure of the Thesis 5

2 Background and Problem Statement **7**

2.1 Introduction 7

2.2 Analysis of the Problem Domain 7

2.2.1 Specification and Realization of NFCs 7

2.2.2 Different Views on Web Services 8

2.2.3 Different Types of Composition 10

2.2.4 Non-Functional Metamodel 12

2.2.5 Middleware for Web Services 15

2.3 Research Project PREMIUM|Services 17

2.4 Related Work 19

2.4.1 NFC Specification 19

2.4.2 NFC Realization and Enforcement 22

2.5 Requirements 27

2.5.1 Specification Requirements 27

2.5.2 Realization and Enforcement Requirements 33

2.5.3 General Requirements 35

2.6 Evaluation of Related Work 36

2.6.1 NFC Specification 36

2.6.2 NFC Realization 40

2.6.3 General Requirements 43

2.7 Conclusion 43

3 NFComp **45**
3.1 Introduction . . . 45
3.2 NFC Composition in Component-based Software Applications . . . 45
3.2.1 Requirements Specification . . . 46
3.2.2 Action Definition . . . 48
3.2.3 Action Composition . . . 53
3.2.4 Action to Application Mapping . . . 56
3.2.5 Action Realization Mapping . . . 57
3.2.6 Code Generation . . . 58
3.3 Towards Conflict-Free Action Compositions . . . 58
3.3.1 Formalizing the Interdependency Model . . . 58
3.3.2 Supporting Action Composition by Validation, Solution Strategies and Conflict-Free Composition Procedure . . . 61
3.4 NFC Composition in a Black Box View of Web Services . . . 66
3.4.1 Requirements Specification . . . 67
3.4.2 Action Definition . . . 68
3.4.3 Action Composition . . . 71
3.4.4 Action to Service Mapping . . . 75
3.4.5 Action to Middleware Service Mapping . . . 78
3.4.6 Generation of NFC Enforcement Code . . . 81
3.5 NFC Composition in a Gray Box View of Web Services . . . 90
3.5.1 Requirements Specification . . . 90
3.5.2 Action Definition . . . 92
3.5.3 Action Composition . . . 94
3.5.4 Action to Composite Service Mapping . . . 96
3.5.5 Scope-Aware Validation . . . 100
3.5.6 Action to Middleware Service Mapping . . . 111
3.5.7 Generation of NFC Enforcement Code . . . 112
3.6 NFC Composition in a White Box View of Web Services . . . 121
3.6.1 Generation of Behavioral Model from Code . . . 122
3.6.2 Action to Service Mapping . . . 126
3.6.3 Action to Middleware Service Mapping . . . 130
3.6.4 Generation of NFC Enforcement Code . . . 131
3.7 Conclusion . . . 134

4 Evaluation **137**
4.1 Introduction . . . 137
4.2 Revisiting the Requirements . . . 138
4.2.1 Specification Requirements . . . 138
4.2.2 Realization and Enforcement Requirements . . . 140

4.3 Criteria-Based Evaluation . . . 142
4.3.1 Scope of Application . . . 143
4.3.2 Specification Complexity . . . 144
4.3.3 Standards Compliance . . . 145
4.3.4 Extensibility . . . 146
4.3.5 Correctness of the Specification . . . 148
4.3.6 Expressiveness of the Composition Language . . . 148
4.3.7 Completeness . . . 150
4.3.8 Separation of Concerns . . . 150
4.3.9 Support for Multiple Users . . . 153
4.3.10 Support for Multiple Roles . . . 153
4.3.11 Support for Multiple Views . . . 153
4.3.12 Support for Multiple Execution Environments . . . 154
4.3.13 Summary . . . 155
4.4 Case Study and Feature Comparison . . . 156
4.4.1 Requirements Specification . . . 158
4.4.2 Action Definition . . . 170
4.4.3 Action Composition . . . 178
4.4.4 Action to Application Mapping . . . 187
4.4.5 Middleware Mapping/Code Generation . . . 196
4.4.6 Summary . . . 209
4.5 Limitations . . . 210
4.5.1 General Limitations . . . 210
4.5.2 Limitations of the Current Implementation . . . 211
4.6 Conclusion . . . 213

5 Summary and Future Work 215
5.1 Summary . . . 215
5.2 Future Work . . . 217
5.2.1 Addressing Limitations . . . 217
5.2.2 Widening Scope . . . 218

A Curriculum Vitae 221

Bibliography 223

List of Tables

2.1 Specification Requirements 33
2.2 Enforcement Requirements 35
2.3 Evaluation of Work on NFC Specification (Black Box View) 38
2.4 Evaluation of Work on NFC Specification (Gray Box View) 40
2.5 Evaluation of NFC Realization Work (Black Box View) 41
2.6 Evaluation of Work on NFC Realization (Gray Box View) 43

3.1 Conflict Matrix for Impact Types 60
3.2 Resolution Strategies: Interdependency Conflicts Solved and Introduced 65
3.3 Actions and Their Impact 69
3.4 Explicitly Defined Interdependencies 70
3.5 Implicit Interdependencies 71
3.6 Subjects of NFAs in the Black Box View 76
3.7 Context Variables Available in the Black Box View 79
3.8 Configuration Entries for the Accounting Middleware Service 81
3.9 Metamodels Used for Code Generation 84
3.10 Subjects of NFAs in the Gray Box View 97
3.11 Overview on Concepts Used in the Formal Specification 106
3.12 Context Variables Available in the Gray Box View 112
3.13 Mapping Java to BPMN2 123
3.14 Subjects of NFAs in the White Box View 128
3.15 Mapping Options in the White Box View 130
3.16 Context Variables Available in the White Box View 131

4.1 Specification Requirements Support by NFComp 140
4.2 Enforcement Requirements Support by NFComp 142
4.3 Metrics for Graphically Represented Metamodel Concepts 144
4.4 Workflow and Dataflow Patterns Supported by NFComp and Other Approaches 149

4.5 When to Use Which View for Web Services 154
4.6 Overview of Criteria-Based Evaluation of NFComp 155
4.7 Feature Comparison in the Requirements Phase 170
4.8 Properties of the Encrypt Action 171
4.9 Feature Comparison in the Action Definition Phase 178
4.10 Feature Comparison in the Action Composition Phase 187
4.11 Feature Comparison in the Mapping Phase 196
4.12 Feature Comparison in the Code Generation Phase 209
4.13 Overall Feature Points Comparison 210
4.14 Performance Reduction Through the Use of Proxies 212

List of Figures

2.1 Specification by Model, Realization by Code Executed at Runtime 8
2.2 Different Views on Web Services: Black, Gray and White Box 9
2.3 Multidimensional NFC Composition from Black Box (Left) and Gray Box Perspective (Right) 11
2.4 Superimposing Concerns Must Be Orchestrated 12
2.5 Conceptual Metamodel for Non-Functional (Light Gray) and Functional (Dark Gray) Concepts 13
2.6 Object Diagram Showing an Instance of the Metamodel 13
2.7 WS Subjects Identified in Eclipse WSDL Editor 29
2.8 Composite Service Subjects Exemplified by the BPMN Process 31
2.9 Control Flow of Composite NFAs Exemplified by Secure Conversation 32

3.1 Approach in a Nutshell: Phases and Involved Roles 46
3.2 The Requirements Metamodel 47
3.3 Requirements Notation 48
3.4 Metamodel of Actions and Properties 50
3.5 Metamodel of Interdependencies 52
3.6 Action Notation 53
3.7 Metamodel of Non-functional Activities 54
3.8 Notation for Control Flow with Non-Functional Tasks 55
3.9 Notation for Data Flow Between Non-Functional Tasks 55
3.10 Notation for a Non-Functional-Activity 55
3.11 Metamodel for the Application Mapping Model 56
3.12 Metamodel for the Action Realization Mapping 57
3.13 Example BPMN for Validation 63
3.14 NFComp Process for Web Services 66
3.15 The Requirements Editor: Modeling Concerns and Attributes 67

3.16 The Action Editor: A Security Expert Has Modeled the Actions and Their Properties and Interdependencies . . . 69
3.17 A Performance Expert Has Modeled Her Actions and Imported a Security Action for the Specification of Cross-Domain Interdependencies . . . 70
3.18 The Composition Editor: Guided Composition Proposes Next Valid Actions . . 72
3.19 The Composition Editor: Conflict Detection and Resolution . . . 73
3.20 The Composition Editor: Resulting Non-Functional Activities . . . 74
3.21 The Extended Mapping Metamodel for Services from the Black Box View . . . 75
3.22 Mapping Notation for Services and Actions . . . 77
3.23 Mapping Notation for Operations and Actions . . . 77
3.24 The Mapping Editor: Mapping NFAs to Web Services . . . 78
3.25 The Extended Action Metamodel for Middleware Mapping Support . . . 79
3.26 The Action Editor: Configuration of the Middleware Service Mapping . . . 80
3.27 Generating Code Out of the Models . . . 81
3.28 Runtime Enforcement of the Modeled NFCs . . . 82
3.29 Generator Components . . . 85
3.30 The Purchase Order Process . . . 91
3.31 The Extended Metamodel for Action Composition in the Gray Box View . . . 94
3.32 The Monitoring Composite Action . . . 96
3.33 The Extended Metamodel for Action Composition in the Gray Box View . . . 97
3.34 Mapping Notation for the Gray Box View . . . 98
3.35 Mapping of Actions and Activities to the Purchase Order Process . . . 99
3.36 Visualization of Process, Events and Scopes . . . 100
3.37 Process Validation Presented in Eclipse-based Mapping Editor . . . 102
3.38 Scopes and Interdependencies Rendered as Sets . . . 109
3.39 Runtime Enforcement of the Modeled NFCs . . . 113
3.40 The Purchase Order BPEL Process Shown in the Eclipse BPEL Editor . . . 117
3.41 The Purchase Order BPEL Process After the Transformation . . . 118
3.42 NFComp Process for Web Services from the White Box View . . . 122
3.43 The Generation Process of Java Code to BPMN2 . . . 124
3.44 Selection of the Relevant Methods and Classes . . . 126
3.45 Generated Behavioral BPMN2 Diagram . . . 127
3.46 Activity Definition for Caching . . . 128
3.47 Mapping of Caching Actions in the White Box View . . . 129
3.48 The Extended Middleware Mapping Metamodel for the White Box View . . . 130

4.1 Separation of Concerns in NFComp . . . 151
4.2 WSDL of the Purchase Order Service . . . 157
4.3 Classes Implementing the Purchase Order Process . . . 158
4.4 Process Logic of the Purchase Order Service . . . 159

4.5 Process Logic Reverse-Engineered From Purchase Order Java Implementation . 160
4.6 NFComp Requirements Model for Purchase Order 162
4.7 Requirements Modeled in the NFR Framework 163
4.8 Requirements Modeled in Sec-MoSC [88] 164
4.9 A Subset of Actions for the Purchase Order Service 171
4.10 Extra-Functional Property Stereotypes for Purchase Order Service in the Approach of Ortiz and Hernandez 174
4.11 Non-Functional Activities for Purchase Order 178
4.12 Composite Actions for Purchase Order 179
4.13 Interdependencies Inferred from Composite Actions 180
4.14 Groups of Actions for the Purchase Order Process with Sec-MoSC 182
4.15 Mapping of Actions to Purchase Order Service (Black Box View) 188
4.16 Mapping of Actions to Purchase Order Service (Gray Box View) 190
4.17 Mapping of Actions to Purchase Order Service (White Box View) 191
4.18 Mapping of Extra-Functional Properties to Purchase Order Service in [68] . . . 192

Chapter 1

Introduction

1.1 Thesis Outline

A software can be regarded logically and physically as a composition of distinct concerns. Concerns are goals or objectives of a piece of software [37]. They focus either on business functionality or on quality aspects of a given software. The former type of concern is referred to as functional or core concern, whereas the latter is referred to as non-functional concern (NFC). Functional concerns are often covered by the main language constructs of an underlying programming language, for example methods and classes in the case of object-oriented languages, whereas for non-functional ones there are usually no corresponding and adequate concepts. This makes non-functional concerns hard to modularize and often results in tangling and scattering, because they crosscut different parts of the software.

To address this problem, a variety of aspect-oriented extensions have been proposed which have already reached wide industry adoption. These extensions allow the modularization of crosscutting concerns into aspects which can be woven with the code representing the core concerns. This weaving mechanism is applied through the use of a query language, called *pointcut*, to select points in the execution of a program which does not require the change of the target code. This approach works quite well for the implementation of non-functional concerns.

However, reality showed that several of those non-functional concerns may superimpose [51] at one functional point; i.e., they must be executed concurrently. This would be no problem, when those concerns were completely orthogonal to one another [61]. This is, however, not the case (cf., for example, Pulvermueller et al. [71] and Sanen et al. [76]), and thus, there are interdependencies or interactions between different concerns which introduce constraints such as mutual exclusion, dependencies and ordering restrictions, among others. This problem is similar to the feature interaction problem originally observed in the telecommunications domain. The composition of several concerns at a certain functional point is not well covered by most aspect-oriented approaches, and a lot of research has been going on in this area for the last couple of years (for example, Shakers and Peters [82], Nagy Nagy et al. [61], Katz [50]

and Durr et al. [28, 29]). Another problem, found by U. Zdun, is that "many software systems suffer from missing support for behavioral (runtime) composition ..." and "The concern 'behavioral composition ...' is not treated as a first-class entity, but instead is hard-coded in different programming styles, leading to tangled composition ... code that is hard to understand and maintain" [102]. Especially runtime adaptation of the composition logic is very limited with this style of programming.

In the last decade, process-oriented approaches evolved to sufficient maturity and thus industry adoption. Those approaches achieved success especially in the context of service-oriented architectures (SOA). Object-oriented languages focus on structural aspects of software, whereas process-oriented languages concentrate on a software's behavioral composition. In process-orientation, the composition logic is made a first-class entity which allows the modeling, analyzing, executing, runtime monitoring and adaptation of business processes. This enables the involvement of non-technical domain experts such as insurance experts or bank officers, because processes can also be understood without too much technical background. A process comprises multiple activities which are executed in a particular order. Each activity performs a specific unit of work such as the invocation of a service. A service implements atomic business functionality. A process can be implemented as a composite service involving multiple partner services to offer a higher level of functionality.

An important insight gained in this thesis is that the process-oriented paradigm is not only valuable for functional concerns (such as implemented in business web services), but can also be applied to non-functional ones such as security, reliable messaging or transactions. The behavior for these non-functional concerns follows well-defined processes, e.g., transactions could be described in terms of the activities begin, participate, commit and rollback. Furthermore, the composition logic of superimposing concerns could also be defined in terms of processes prescribing when which behavior should be executed. In current web service implementations, the composition logic of non-functional concerns is completely hidden from the user. For example, the declarative policy standard for web services — WS-Policy [95] — that is used to describe which non-functional behavior must be enforced for which service, does not support ordering concepts appropriately. The order is statically implemented and cannot be adapted easily.

One of the aims of this Ph.D. work is thus to provide a methodology to adopt process-oriented concepts for the composition of non-functional concerns. This allows the closing of severe gaps in the state of the art which currently prohibit the flexible composition of non-functional concerns at runtime. To address both specification and realization/enforcement issues, a model-driven approach has been chosen which allows the intuitive graphical modeling of processes and the execution of the processes by generating enforcement code. Furthermore, there are different views on web services [4] — black, gray and white box — which must be considered in this context. In the black box view, only the WSDL [17] of the service is visible. In the gray box view, the composition logic is available in addition to the WSDL, and in the white box view all implementation artifacts are visible. Depending on the view, different types

and complexities of non-functional concern compositions can be supported. The support is more powerful when moving from black over gray to white box view. However, the dependencies also increase with the number of visible artifacts, making all three views valuable for the composition of web services with non-functional concerns. This is why the holistic approach described in this thesis addresses and supports all three views. However, the emphasis is more on the black and gray box view, because they are more lightweight with respect to implementation dependencies.

1.2 Contributions

The following paragraph presents the thesis of this work.

Expressive composition languages for non-functional concerns in service-oriented architectures are needed, capable of expressing control and data flow interactions within and across non-functional concerns.
The work justifies the need for such languages by a comprehensive analysis of the state of the art, presents the design and implementation of such a language and validates it.

More specifically, the following contributions are provided by this work:

- *Requirements identification*
  This Ph.D. work identifies the first detailed set of requirements for both, specification AND realization/enforcement of non-functional concerns in web services.
- *State of the art evaluation*
  This Ph.D. work presents a detailed evaluation of state-of-the-art approaches in terms of requirements fulfillment. Approaches for different types of web services have been analyzed, specifically atomic web services as well as composite ones.
- *Generic approach for fine-grained composition of non-functional actions*
  The novel NFComp approach proposed in this work allows the composition of fine-grained non-functional actions. This allows a flexible and dynamic composition specific to the concrete scenario/service. The approach is generic enough to be applied to all kinds of non-functional concerns. It is also the first work that distinguishes and supports horizontal and vertical composition, and composite and atomic non-functional actions.
  - *Development process with different roles and phases*
    NFComp proposes a development process involving different roles and phases to facilitate the application of the approach, also in enterprise contexts, in which a variety of people with different responsibilities will collaborate in order to cope with complexity and to achieve an optimal result.
  - *Metamodel for non-functional concerns*
    The model-driven methodology used in this approach relies on a general metamodel.

This metamodel is not only the abstract syntax for the underlying models but can also be used to gain a deep understanding of the involved non-functional concepts. Furthermore, the metamodel is platform-independent and allows the construction of different tool sets.

- *Graphical notation for the composition of NFCs*
  NFComp includes a lightweight, concrete graphical syntax which is based on BPMN2 complemented by a set of custom notations. This graphical notation is intuitive, and only few new concepts must be learned.
- *Tool set for editing models and code generation*
  This work does not only include a novel concept for concern composition, but also an implementation in terms of Eclipse[1]-based modeling editors and a code generator that turns the model into executable code.
- *Design time validation of the models*
  The models created with the tool set can be validated at design time to find possible conflicts as early as possible.
- *Strong focus on separation of concerns*
  NFComp encourages and strongly supports the separation of concerns principle. It allows the separation on the modeling as well as on the code level. Each concern is modularized into its own component, whether functional or non-functional. Even the composition concern is separated in its own model and a dedicated runtime component is responsible for its enforcement. This fosters understandability, reuse and maintainability.

- *Architecture and implementation preserving web services' inherent requirements*
  This Ph.D. thesis proposes a runtime architecture based on proxies to enforce the modeled behavior. This architecture preserves the distribution and platform-independence of web services and allows the integration of new non-functional concerns at runtime without changing the web service's implementation.

The following papers were published in the context of this Ph.D. thesis:

1. B. Schmeling, A. Charfi, M. Martin, and M. Mezini, "Towards Conflict-Free Composition of Non-functional Concerns," in *24th International Conference on Advanced Information Systems Engineering (CAiSE'12)*. Gdansk, Poland: Springer, June 2012.

2. B. Schmeling, A. Charfi, S. Heinzl, and M. Mezini, "A survey on non-functional concerns in web services," *International Journal of Web Information Systems (IJWIS)*, vol. 8, no. 1, pp. 5–31, 2012.

[1] `http://www.eclipse.org`

3. B. Schmeling, A. Charfi, R. Thome, and M. Mezini, "Composing Non-Functional Concerns in Web Services," in *The 9th European Conference on Web Services (ECOWS'11)*. Lugano, Switzerland: IEEE Computer Society, September 2011.

4. B. Schmeling, A. Charfi, and M. Mezini, "Composing Non-Functional Concerns in Composite Web Services," in *IEEE International Conference on Web Services (ICWS'11)*. Washington DC, USA: IEEE Computer Society, July 2011.

5. B. Schmeling, A. Charfi, and M. Mezini, "Non-functional Concerns in Web Services: Requirements and State of the Art Analysis," in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, Paris, France, November 2010.

1.3 Structure of the Thesis

This thesis is structured into five chapters. The current chapter, ***Introduction***, provides a brief presentation of the research topic, delivers a condensed overview on all contributions and finally elaborates on the structure of the thesis. The second chapter, ***Background and Problem Statement***, systematically analyzes state-of-the-art approaches for the composition of non-functional concerns in web services. It presents a list of the most important requirements motivated by the investigated related work and the research project PREMIUM|Services. Finally, the most important, not yet addressed, requirements are outlined. In the third chapter, ***NFComp: Modeling Executable Non-Functional Action Compositions***, a novel approach for addressing the open research problems and requirements — called NFComp — is presented. This approach is model-driven, it allows the modeling of the concern composition and generates executable code out of these models. The models are represented graphically and can be modeled and validated using a set of integrated editors. The fourth chapter, ***Evaluation***, provides an assessment of the quality of the approach. This quality is measured in terms of requirements fulfillment, satisfaction of general high-level criteria for software engineering approaches and a feature comparison. Furthermore a case study is presented which illustrates the applicability of NFComp and shows how it compares to other approaches. The fifth chapter, ***Summary and Future Work***, summarizes the contributions and reviews the results of this thesis while outlining prospects for future work.

Chapter 2

Background and Problem Statement

2.1 Introduction

In this chapter, the research problem being addressed by this thesis is presented and motivated. Firstly, a brief overview of the problem domain is given. Then, the research project which initially justified the research in this problem domain is presented. Subsequently, a detailed analysis of related work is conducted in order to find already existing approaches capable of supporting these scenarios and to provide an overview on the state of the art in general. Furthermore, based on the capabilities of state-of-the-art approaches and based on the scenarios from the research project, a set of requirements is derived. The state-of-the-art approaches are then evaluated against these requirements. Finally, a conclusion is given outlining the most relevant research problems that should be addressed; firstly, to support the scenarios and, secondly, to compensate the drawbacks of existing approaches. Parts of this chapter have been published in [79, 80, 81, 77].

2.2 Analysis of the Problem Domain

In this section an overview and analysis of the problem domain is given. More specifically, the specification and realization aspect of NFCs is explained, different views on web services are exhibited, different types of composition are motivated and, finally, an abstract, conceptual metamodel is introduced. This metamodel visualizes the domain of non-functional concerns and its corresponding concepts in the context of web services. Furthermore, the most important terms used throughout this thesis are defined, and the role of middleware is discussed. The section concludes with a brief discussion about middleware for web services.

2.2.1 Specification and Realization of NFCs

The support for non-functional concerns encompasses two aspects: the specification of NFCs on the one hand and the realization of the specified NFCs on the other hand. This is a general

characteristic of methodologies that are based on models. A model takes a particular view on a problem domain such as NFC composition and creates an abstraction of the often more complicated real world. In this regard the model may describe the underlying software artifacts such as the source code, the data base structure or the interfaces of components. Since NFCs tend to be crosscutting and thus scattered around many software artifacts, it should be an objective of the NFC model to modularize these concerns and focus on a particular aspect such as the behavior or the structure of the concerns and the functional subjects they belong to. However, the model also should be enforced at runtime; i.e., if the behavior of NFCs is described in the model, then this behavior must be exhibited by the software at runtime. A model that can directly be executed is called an executable model (cf. for example Executable UML [57]). However, a model can also be made indirectly executable, e.g., by transforming it into executable code or into runtime artifacts that are responsible for enforcing the modeled behavior. The advantage of the indirectly executable model is that the developers can access and modify the generated code artifacts and complement the possibly missing information later, for example, because particular information has been omitted in the model intentionally for abstraction purposes. However, since there are two layers, the model and the code layer, the problem of modularizing crosscutting concerns should be solved for both the aspects of specification and realization.

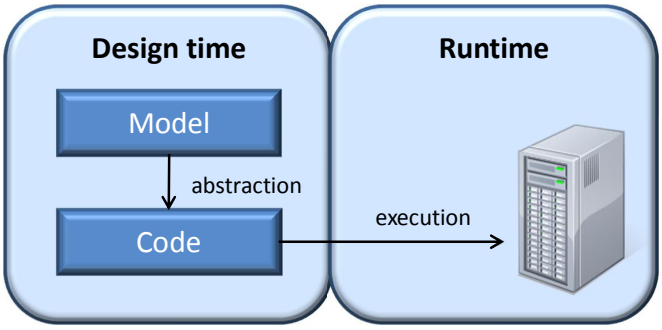

Figure 2.1: Specification by Model, Realization by Code Executed at Runtime

Figure 2.1 shows specification and realization for NFCs. At design time NFCs are specified in a model which is transformed into executable code. The code can then be adapted and complemented by developers and executed on a particular machine at runtime. The behavior of the software should conform to the specification in the model. However, some models are too abstract and cannot be executed or be transformed into code. These kinds of models can only be used as documentation or a design template. A developer can then manually write the code according to the documentation. However, since a manual step is involved, the model and the code could get out of sync, resulting in runtime behavior which does not conform to the model.

2.2.2 Different Views on Web Services

Web services can be regarded from different views: black box view (i.e., only the service interface is visible), gray box view (i.e., the internal process of a composite service is visible in

addition to its interface), white box view (i.e., all implementation artifacts are visible) and glass box view (i.e., purely declarative view on the service, describes what the service does but not how) [4]. In this thesis, the focus lies on the black, gray and white box view because they contain essential information about the behavior of a web service, which is important for a behavioral composition of non-functional concerns. The glass box, however, does not describe any of these details. Nonetheless, the different views can be interpreted differently.

In this thesis, the views define a spectrum from black to white with different levels of gray in between. On one side of the spectrum, black defines that there is only a minimum of information available for this service (its interface). On the other side of the spectrum, white indicates that everything is known about the service (including its source code). The gray box lies somewhere between black and white: More information is available than in the black but less information than in the white box view. Theoretically, the gray color of the gray box may have different shades: a darker one with only a small amount of additional information compared to the black box, or a brighter one with much more information than in the black box but less information than in the white box. In this thesis, gray box means that the internal composition logic is available in addition to the service's interface.

Figure 2.2 depicts the different views. The black box view contains information about the service itself, its operations and the structure of the messages being sent or received by the service. This information is stored in the WSDL of the service. The gray box additionally contains information about the composition logic in terms of process logic; i.e., when, which partner service is called by the service, described by a workflow language such as BPMN [67] or WS-BPEL [2]. The white box provides all programming artifacts available for the service, for example the WSDL, and the implementing source code (which will also contain the composition logic, but most probably it will not be as explicitly as it is described in BPMN).

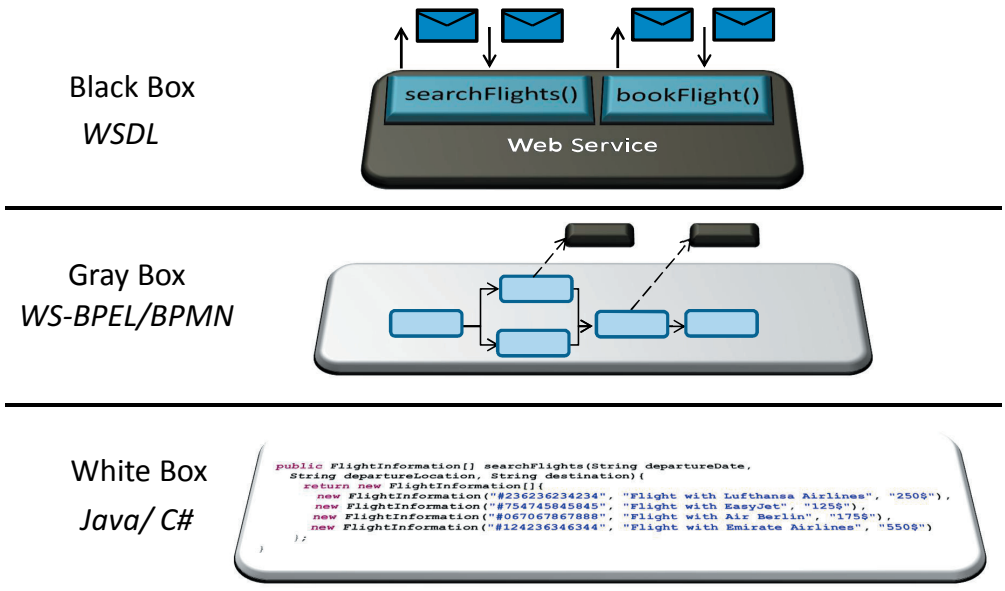

Figure 2.2: Different Views on Web Services: Black, Gray and White Box

The different views are used in the context of this thesis because they systematically define discrete levels of accessible information. On the basis of this information, an analysis can be done of what can be achieved towards composition of non-functional concerns when this information is available. It is assumed that the more information available, the more fine-grained and powerful the composition of non-functional concerns will be. However, the more information being used by the approach, the higher the dependencies are. For example a black box solution can be used for all kinds of web services (all have a WSDL interface). The gray box can be used for composite web services described in BPMN or WS-BPEL, but not for atomic ones, and the white box can be used for all types of services but may introduce dependencies to the programming language being used for the service's implementation, e.g., Java or .Net.

2.2.3 Different Types of Composition

Figure 2.3 shows the NFC composition from the black box and gray box view. The layer at the bottom represents the core concern which is complemented by additional NFCs, each rendered as a separate layer. In each layer there are non-functional actions represented by rounded rectangles that contribute behavior to the respective concern. Each action is woven with a functional component represented by the dashed lines connecting the core layer with the NFC layers. On the left side of the figure the black box view is taken showing only punctual, atomic actions woven with web services. The right side of the figure shows the gray box view in which web services expose their process logic and also NFAs in the same concern layer are considered processes.

Furthermore, there are often multiple NFAs that must be applied to the same point of execution which are marked as gray ovals with an S in Figure 2.3. The NFCs superimpose at these points and because of interactions between different non-orthogonal NFAs (NFAs which affect each other or interact with each other) an execution order must be defined explicitly. Otherwise the resulting behavior/impact of the superimposed NFAs cannot be anticipated. Figure 2.4 shows an example for defining the execution order of the three NFAs *Encrypt*, *Sign* and *Accounting* explicitly. In summary, there are three different composition types to be distinguished:

- **Composition of functional concerns with NFAs** This is often referred to as *weaving* or *instrumentation* of the functional code with non-functional code. There must be a well-defined set of execution points in (composite) web services that must be woven with NFAs.
- **Composition of superimposing NFAs** This can be referred to as a *vertical composition* because the composition happens at one execution point where more than one NFA is executed at once.
- **Composition of composite NFAs** This can be referred to as *horizontal composition* since NFAs are composed in a similar way as the functional business processes and hence called composite NFAs.

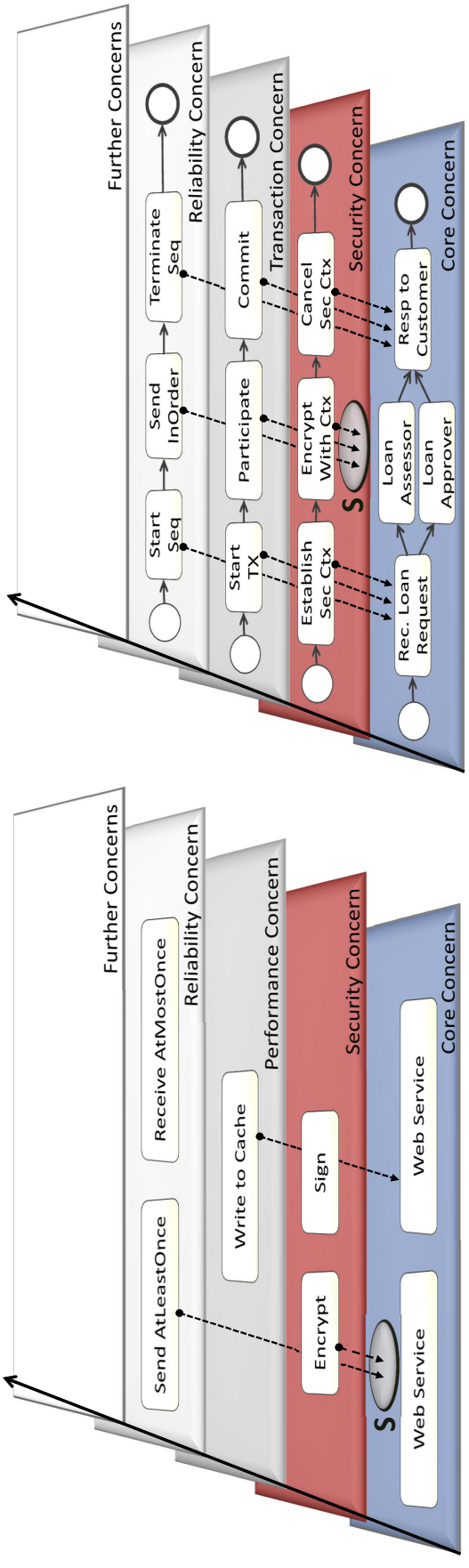

Figure 2.3: Multidimensional NFC Composition from Black Box (Left) and Gray Box Perspective (Right)

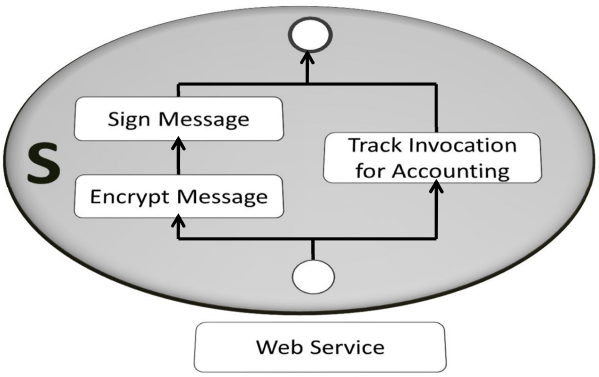

Figure 2.4: Superimposing Concerns Must Be Orchestrated

With composite web services and the three different composition types there are complex dependencies between services and NFAs along different dimensions (as can already be observed in Figure 2.3). While a graphical approach is complex enough although abstracting away most of the details (such as the implementation of NFAs or the structure of SOAP [63] messages), a pure code-based approach would be much more complex and would hide the composition logic behind implementation details. Moreover, the composition logic would be tangled and scattered throughout a set of services. The composition logic should rather be directly visible, adaptable and hence modeled explicitly. Otherwise the resulting software system would obtain modularity but the composition behavior of interacting stateful NFAs and the underlying business processes would be unclear. With a well-defined modeling language the complexity could be overcome and the code responsible for the composition could be generated out of the model. This would allow adapting the composition logic and hence the resulting non-functional behavior without changing the underlying composite web services. The application of process-oriented modeling of functional behavior especially in SOAs has turned out to be very successful recently. This knowledge could also be applied to the composition of non-functional behavior.

2.2.4 Non-Functional Metamodel

In the following a brief explanation of the most important terms and concepts that are used throughout this thesis are given. The concepts and their relations are shown in a simplified conceptual model depicted in Figure 2.5. The depicted diagram is an EMF[1] Ecore diagram which is an implementation of the EMOF [65] standard for metamodeling. The syntax is similar to that of UML class diagrams. A rectangle represents an *EClass* which has two compartments, one for attributes and one for operations, which have been mostly omitted in this conceptual model. Connection lines between *EClasses* represent references from one class to another. *EReferences* come in two flavors: pure references and containment references. The containment reference is similar to the composite association in UML and binds the life time of the child (the element opposite to the black diamond) to that of its parent.

[1] `http://www.eclipse.org/modeling/emf`

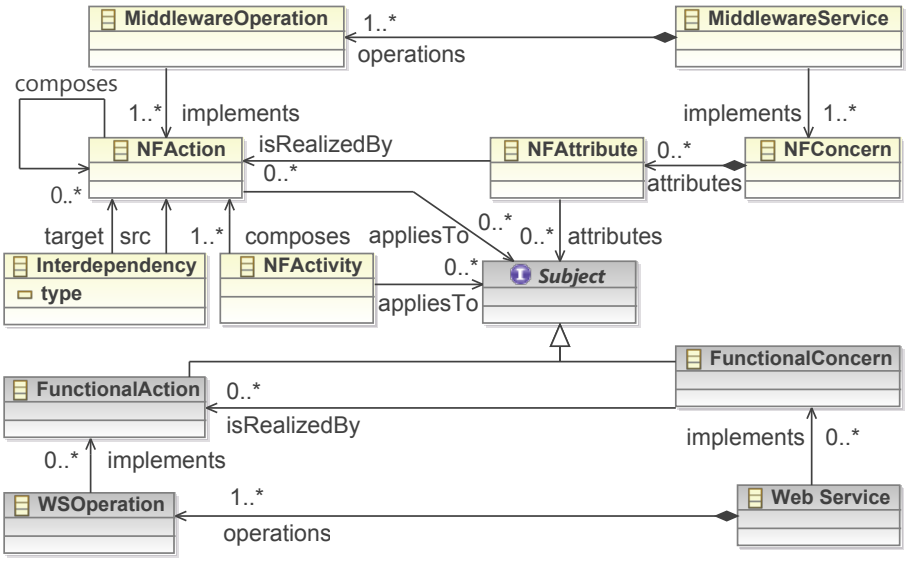

Figure 2.5: Conceptual Metamodel for Non-Functional (Light Gray) and Functional (Dark Gray) Concepts

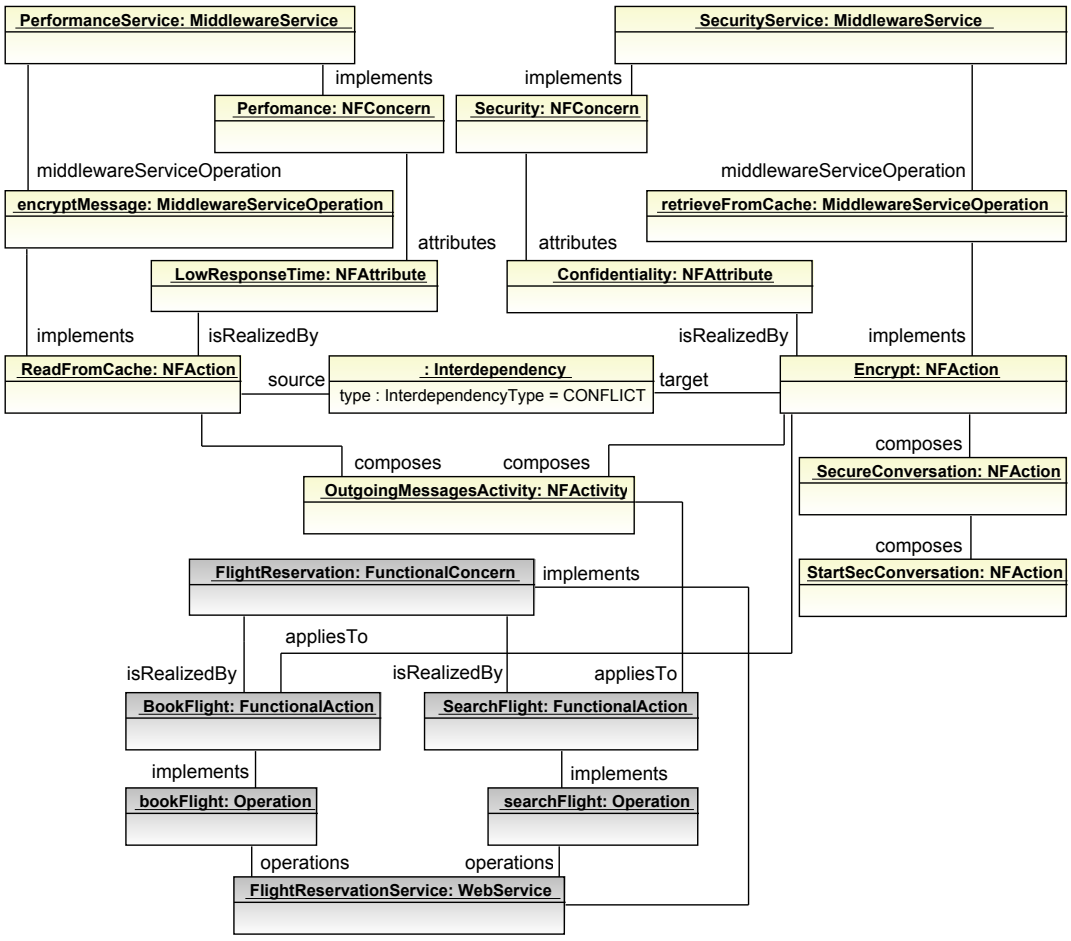

Figure 2.6: Object Diagram Showing an Instance of the Metamodel

Figure 2.6 shows an illustrative example for an instance of the metamodel using a UML Object Diagram [66]. The object diagram shows instances of classes by rectangles with one compartment. An instance has a name and a type (corresponding to one of the meta classes). Instances of references are shown as connection lines. In the following the visualized concepts and relations of the metamodel are described in more detail using the instances of the object diagram as examples.

- **Functional concern (FC):** A coherent piece of functionality that is part of a software system, for example a concern *FlightReservation* for a travel application. An FC can be described as a use case specified in the system requirements. In a perfectly decomposed software system a web service will realize exactly one FC which makes the service coherent.

- **Functional action (FA):** Behavior such as *BookFlight* or *SearchFlight* that can be executed in order to realize (parts of) the more coarse-grained FC. One FC is constituted of multiple functional actions. From the granularity point of view an FA is — assuming again an ideally decomposed system — realized by one operation of a web service. The abstraction layer for FCs and FAs is introduced because web services are only one possible technology that can be used to realize the concerns.

- **Non-functional concern (NFC):** Non-functional Concern is a general term describing a matter of interest or importance that does not correspond to a functional but a non-functional matter of a system. An NFC may involve both the requirements and the implementation of the requirements. In this metamodel the term NFC is used for requirements definitions thus representing a group of semantically correlated non-functional attributes. *Security* and *Performance* are examples for NFCs.

- **Non-functional attribute:** An attribute that describes the quality or characteristics of a non-functional concern. For example, *Confidentiality* would be a non-functional attribute for an NFC *Security* which is required for an FC *FlightReservation*.

- **Non-functional action (NFA):** A non-functional action is a distinct, abstract and reusable element that represents behavior affecting non-functional attributes (cf. Rosa et al. [73]). The behavior represented by an NFA is assumed to be realized by operations exposed by a particular service or software module, for instance a middleware service. An NFA is only executable in a functional context and may consume data from this context or produce and change data itself. NFAs can be atomic or composite. Composite NFAs are constituted by several atomic NFAs which are executed in a particular order (horizontal composition). This allows the definition of NFAs on different abstraction levels.

- **Middleware Service (MWS):** A middleware service implements non-functional behavior corresponding to a specific NFC (ideally one, but could be more than one). It exposes

a set of operations, each implementing the behavior of a specific NFA. There are two different flavors of MWS: stateful and stateless ones.

- **Subject of an NFA:** A well-defined point that an NFA or activity applies to. Examples are FCs or particular FAs like *SearchFlight* or *BookFlight*.

- **Non-functional activity:** A complex order or control flow of NFAs that apply to the same subject (vertical composition, for example a sequence of two actions *Decrypt* and *ReadFromCache*). A non-functional activity (the terms are used in analogy to activity and action in UML2 [66]) is — like NFAs — neither visible nor directly consumable by the service consumer.

- **Interdependency:** An interdependency describes interactions between actions and imposes constraints with respect to the composability of actions, ordering restrictions, among others. An interdependency is a binary relationship between two actions.

2.2.5 Middleware for Web Services

A middleware is a software layer located between operation system and application layer. It offers several services related to heterogeneity of the underlying infrastructure and distribution/interoperability of applications among others (c.f. the definition of ObjectWeb[2]). The offered services could be coarse-grained such as a security service or fine-grained such as an encryption service. The implementation of such services could be modularized or scattered and tangled in the middleware system.

In the context of web services, there are various aspects for which middleware plays an important role. Firstly, middleware offers additional services to enhance the internal implementation of a web service. A JEE sever is an example for such a middleware. It provides a set of services for security, transactions, persistence and messaging to applications. In this case, middleware services are offered as programming libraries to be consumed via API calls or annotations in the code. Secondly, middleware offers services on the messaging level, i.e. at the SOAP level. In this case, a middleware service usually implements a particular WS-* specification. It is often integrated via handler mechanisms in the SOAP framework or using a central messaging component such as an enterprise service bus.

Similar to web services, middleware can be distributed as well. Middleware can be split into multiple parts residing at different sides of the communicating participants. In particular cases (for instance, for access control) it is sufficient to have a middleware only on one specific side, e.g., the web service provider side. However, in other cases, a particular feature can only be implemented when middleware is distributed to the consumer and provider side. An example for this is reliable messaging. In reliable messaging, four types of message delivery assurances are supported: *at most once*, *at least once*, *exactly once* and *in order* [26]. The *at least once*

[2] `http://middleware.objectweb.org`

assurance can be enforced by a reliable messaging endpoint (middleware service) at the receiver side, which sends acknowledgements to the sender whenever a message arrives, and one at the sender side, which resends the message if no such acknowledgement arrives after a given time. The *at most once* delivery assurance is similar. The sender must add unique message identifiers whereas the receiver must filter out duplicates based on those identifiers. A third possibility is to have a middleware provided by some third-party intermediary. This allows the support of features that should be implemented independently of consumer and provider of a web service, e.g. monitoring of SLAs. Generally, combinations of the three variants could also make sense; for instance, middleware as intermediary plus middleware on the web service side.

In this work, middleware services are directly connected to non-functional actions: A middleware service implements an NFA. Hence, the actual, available middleware system to be used should be kept in mind when modeling NFAs. An NFA cannot be more fine-grained than the underlying middleware system. That is, when the middleware service delivers security as a whole (for example, including signatures, encryption and authorization), an NFA should not be defined on the level of *Encrypt* because there is no consumable service functionality for this. However, the inverse, defining coarse grained NFAs although more fine-grained middleware functionality is available, can indeed be reasonable. In this case composite actions can be used, e.g. an NFA *Security* but this action should be composed of more fine-grained atomic actions on the same level as the middleware functionality. An assumption made in this thesis is that the model to be executable, must contain only NFAs which can be implemented by corresponding and consumable middleware functionality. The consumability in this respect depends on the view of the web service and the concrete execution environment.

With respect to the distribution of middleware services, there are two potential views of the system. The first one takes only a single component into account; for example, a particular web service, and regards the non-functional requirements for this service. Then, the corresponding NFAs are determined and middleware services are chosen to be integrated into the web service. The second view regards multiple components at once. As discussed above, some NFCs require middleware on consumer as well as provider side to satisfy a particular requirement such as a certain reliable messaging assurance. Hence, particular concerns cannot be implemented for a single component regarded in isolation. The problem with this view is, however, that not all components participating in a service-oriented architecture are offered by the same provider. Furthermore, different providers may use different techniques to enforce the given requirements. Even worse, the non-functional requirements for communication with remote, third-party web services are often more important than the one between local ones. For example, when a message is sent via the Internet to another provider, this message may use different transport protocols and thus it is not clear if the message can be delivered reliably (which is one of the motivations for reliable messaging protocols at the SOAP level, for instance).

Hence, it makes no sense to assume a global view of those web services in general. It is more realistic to adopt the first view, although it may be limited in the consideration of particular NFRs. Adopting this view may result in interoperability issues. If one component requires reliable messaging and the other does not support it, the reliable communication cannot be established. This issue can be addressed by using WS-Policy [95] to describe the capabilities and requirements and match them to find the effective policy before the communication takes place. If the effective policy is empty, the services will be incompatible and thus cannot communicate with one other, or they cannot satisfy the desired NFRs. This aspect of interoperability is, however, not in the focus of this thesis. It concentrates more on the aspect of how to enforce requirements and actions for a particular component. Nonetheless, it defines concepts to mitigate the issue of interoperability by introducing the *inverse* interdependency in Section 3.2.2.2 and shows how this can be leveraged to generate compatible/interoperable consumer and provider logic at the end of Section 3.4.6. However, the prerequisite for this approach is that both consumer and provider side can be accessed — to a certain degree — by the proposed framework/solution.

2.3 Research Project PREMIUM|Services

This section describes the research project which motivated the research in the field of non-functional concern composition presented in this thesis. This thesis has been written in the context of the two publicly funded research projects PREMIUM|Services and InDiNet, whereas PREMIUM|Services mainly drove the investigation of composition of non-functional concerns.

The project PREMIUM|Services[3] is a research project with strong industry participation and also with strong business relevance. It aims at providing commercial web services implementing innovative pricing mechanisms (which have been developed in the project) for small and medium enterprises (SME). These pricing web services are published in the PREMIUM|Services marketplace where B2B customers like online shops can find and purchase subscriptions to the services. In addition, the marketplace offers a set of third-party middleware services (MWS) which implement non-functional actions such as accounting, logging, access control etc. When a service provider publishes her pricing service in the marketplace, she should be able to select from these MWS in order to enable them for her published service. Also customers should be able to make use of the MWS, e.g., to stay interoperable with a security-enhanced pricing service. An example of a pricing service would be a dynamic posted pricing (DPP) service which calculates dynamic prices based on extended customer data, for example purchase history and customer value. This service offers multiple operations: an operation to transmit purchase history, product information etc., an operation to configure the service and an operation to calculate the price for a specific customer.

[3] http://www.premiumservices.research-events.com

The presented scenario introduces a set of requirements that must be fulfilled. As already explained, a service provider considers enhancing her pricing web service with additional NFCs which have been neglected during the — purely functional — service development. For example, the service provider of DPP could have decided to use the Accounting MWS (which tracks service invocations for billing purposes), the Caching MWS (which looks up already calculated results from a cache), and the Encryption MWS (which protects messages from being read by unauthorized persons).

However, it is not clear at this point in which order the MWS should be applied to the service; e.g., should *Accounting* be executed before or after *Caching*? If *Accounting* were executed before *Caching*, then every service invocation would have been accounted for, whereas in the reverse order only those invocations would have been accounted for that were not cached. In the former case the customer must pay for each service invocation whether it is cached or not, whereas in the latter case the customer only has to pay if the message has not already been cached. Each price calculation in DPP requires an inquiry of credit agencies which must be paid. If an online shop has integrated DPP and it is invoked each time a product is queried by a shop user, this would cause high costs. With caching, these costs could be minimized for the provider of DPP and its consumer, i.e., the online shop. Hence, in this scenario, it makes sense to let the service consumer pay only those invocations that were not already cached. However, in other scenarios, the former case may be more appropriate, which shows that specifying the execution order of NFAs (and thus MWS) depends highly on the use case and thus should be specified explicitly, which is an important requirement.

Some NFAs cannot be considered to be completely independently of others, i.e., they are non-orthogonal. For example, the Accounting MWS may require the usage of the Authentication MWS; otherwise the identity of the service consumer cannot be proved, which is certainly required for *Accounting* due to legal issues. Furthermore, there are also other types of dependencies among NFAs, for instance NFAs which are in conflict with each other when executed in the same context. Without a proper dependency specification at design time, the MWS will not behave correctly at runtime. In this scenario the selection of the service provider additionally requires the use of the Authentication MWS. To be aware of this, dependencies between NFAs should be specified.

Assuming the provider would have specified the execution order and the dependencies, the next problem which she is faced with is that the selected MWS must be associated with the functional subject. It is not sufficient simply to apply NFAs to a web service which is far too coarse-grained because particular NFAs only make sense if applied to incoming messages (and not to outgoing) and some are only valuable for specific service operations. In this scenario, *Caching* should only be applied for the price calculation operation and not for the configuration operation (which is one-way). Moreover, *Encryption* should only be applied to outgoing messages, not for incoming ones (which should instead use *Decryption*). Hence, fine-grained service subjects should be specified for NFAs.

After the specification of order, dependencies and subjects the service provider needs an architecture that is capable of realizing the specification at runtime. The realization of NFA specification should be platform-neutral, which means it should completely rely on web service standards and not on the concrete programming language that has been used to implement the web service. Otherwise, the component which realizes the specification cannot be applied to all kinds of services which is a general requirement but especially important in this scenario: Pricing web services have been implemented by different providers using different programming platforms, more specifically Java and .Net.

The software components used for the realization of NFAs must enforce the specification at runtime. This could be implemented in a manual way using the specification as requirements documentation for the software architecture. However, this is not sufficient since it would require too great an effort and would be error-prone, especially when several different specifications must be realized or existing ones must be changed. Using the specification as input for code generation automates the transition from specification to realization by materializing this knowledge into a code generator. Ideally, the code generator should only generate those parts that will change when the input specification changes; i.e., there should be a static code skeleton with a configurable part that can be produced by the code generator.

2.4 Related Work

In this section, related work regarding NFC composition for web services will be identified and described. The purpose of this section is to analyze whether state-of-the-art work can support the described scenarios and to identify general requirements. The approaches are categorized by specification and realization/enforcement and black and gray box. Real white box approaches have not been identified. The reason is, on one hand, that there are approaches assuming a particular programming language and integrating NFCs on this internal level, but, on the other hand, the observation was that these approaches usually do not go beyond the interface level.

2.4.1 NFC Specification

In the following, specification approaches are presented for black and gray box view.

2.4.1.1 Black Box Approaches

There are works in the field of requirements engineering, especially Goal-Oriented Requirements Engineering (GORE [93]) is relevant for NFC composition. Popular representatives of GORE are the NFR Framework [18], i* [30], KAOS [25] among others.

Chung et al. introduced the NFR framework in which so called softgoals (in terms of this thesis, non-functional concerns and attributes) are decomposed and structured into trees with multiple substructures. The different goals and subgoals can interfere, for example security may lower usability.

ProcessNFL [73] is a language that is intended to be used to express non-functional requirements during software development. The language defines three first-class entities; non-functional attributes, non-functional actions and non-functional properties. Non-functional attributes model non-functional characteristics that are quantifiable (for example *performance*), non-quantifiable (e.g., stated informally such as a certain level of security) or being either met or not met (such as *atomicity* or *isolation*). A non-functional attribute can be further decomposed into primitive non-functional attributes. Non-functional actions are defined as software characteristics such as design decisions, algorithms, or data structures, among others, or hardware characteristics concerning computer resources, that affect (realize or have an effect on) non-functional attributes. Non-functional properties model constraints over non-functional attributes, for example the attribute *performance* can be constrained by a property "good performance". In addition, a non-functional property can be assigned a priority in order to decide which non-functional actions should be considered more important.

Soeiro et al. [86] propose a language based on XML that facilitates the specification and composition of concerns at the requirements level. Each concern can be described by properties like its name, classification and stakeholders.

Furthermore, there are works that focus on specification of non-functional actions instead of requirements. The relationship between actions and requirements is that requirements describe the goals to be achieved for a particular software whereas actions define how this can be achieved. An example for a requirement would be *confidentiality*, and the corresponding action supporting this requirement would be *encryption*.

WS-Policy [95] is a declarative approach to define non-functional requirements or capabilities for web services where each non-functional concern is represented by its own domain-specific WS-Policy specification. For example security is covered by the WS-SecurityPolicy [55] specification which defines a set of security related assertions. An assertion represents a requirement, a capability, or another property of a behavior. In case of security, there are—among others—protection assertions that define what is protected at what level; for example, one can specify that the message body should be signed and encrypted. The subject of a policy is defined as an entity (e.g., an endpoint, message, resource, operation) which a policy can be associated with (see WS-PolicyAttachment [96]).

The SCA Policy Framework [5] is based on WS-Policy. In this framework an *intent* describes the non-functional requirements such as *authentication* or *confidentiality* of a component or an interaction between components. The information on how this requirement is realized must be known at deploy time. For example the confidentiality *intent* can be realized by attaching a WS-Policy that specifies encryption assertions.

In some approaches only a single concern is considered. Fox and Jürjens [33] use composition filters at the modeling level in order to specify the composition/weaving order of a security concern with the functional concern.

Sanen et al. [76] define interactions between concerns on a conceptual level. They provide a model where concerns (which represent requirements), components realizing the concerns, and interactions are first-class citizen, and they introduce a classification of interactions: *Assistance*, *Choice*, *Conflict*, *Dependency* and *Mutex*. With this information a set of rules can be generated that supports application development teams to reason over the interaction of middleware concerns.

Another group of approaches uses UML profiles for modeling non-functional attributes. For instance, the UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [40] describes a set of QoS characteristics such as throughput, latency, integrity, and dependability, among others. The characteristics can also be parametrized with, for example, the appropriate type of keys or types of encryption algorithms in case of the security characteristics.

Ortiz and Hernandez [68] incorporate extra-functional properties into UML models by the use of UML stereotypes. For example, there are stereotypes for encryption, decryption and logging which can be added to classes or operations that are representing services. The annotated model can then be used to generate AspectJ code that realizes the specified concerns.

Sanchez et al. [75] introduce a flexible architecture to build extensible workflow applications. In their approach, domains involved in a workflow are identified and implemented as concerns using executable models. These executable models are described as open objects which play the role of coordination elements. An open object is a triple entity (object with attributes and methods), associated state machine (which is an abstraction of the entity's lifecycle) and a set of actions representing behavior that is associated with transitions between states of the state machine. The open objects of different domains can be woven by creating a relationship between the transition of one open object and the action of another open object. The semantics is that the thread of control moves to the second open object, executes the action and causes a transition to another state in its state machine.

Bergmans and Aksit [6] introduce a general model of multi-dimensional concern composition. This model is a combination of a multidimensional concern model as a set of typed graphs and a hierarchical nested model. A concern is represented by an ellipse and can be connected with other concerns via two different kinds of connections. The first connection type represents composition relations to other nested concerns whereas the second type defines dependencies between concerns. Each concern is assigned to a concern dimension represented by dashed lines in their model.

Naveed et al. [62] propose to model aspect behavior (comparable to NFAs) at shared join points using state charts. They use three composite states: one to capture the core requirements and two for the aspectual requirements before and after the core requirements. In each composite state, the order and control of superimposing requirements is described; for instance, if two aspectual requirements should be mapped before a core requirement, the order of those aspectual requirements will be defined in the *before* composite state.

2.4.1.2 Gray Box Approaches

The work of Chollet and Lalanda [16] presents a generative framework to address NFCs for web service compositions. As part of the framework they separate the orchestration metamodel from other non-functional metamodels. A non-functional metamodel for security is provided where concepts exist for expressing requirements and how these requirements are ensured.

The AO4BPEL [14] deployment descriptor allows for specifying non-functional requirements for BPEL processes. First the *service* (one of logging, security, transactions and reliable messaging) is selected. Then the *class* (e.g., authentication or confidentiality for security) can be specified. The *type* (e.g., encryption or decryption for confidentiality) is then the concrete action that must be executed in order to enforce the requirement. Then an XPath[19]-based *selector* can be defined, for instance for a certain BPEL activity. Each *selector* can be mapped to a requirement.

Weber et al. [98] focus on the process level and thus allow for the composition of BPMN processes with so-called delivery components that represent non-functional behavior. The authors describe the lifecycle/process of non-functional behavior in terms of state diagrams that can be composed with functional BPMN processes. The composition is realized by defining a certain state of a composite as a pre- or post-condition of a BPMN task, and the result of the composition is a new BPMN process that contains the transitions (represented by tasks) that are necessary to reach the state that has been specified.

AO4BPMN [13] is an aspect-oriented extension to BPMN supporting the modularization of crosscutting concerns such as compliance, accounting, billing, monitoring, authorization, etc., and provides a visual syntax for AOP concepts like pointcuts and advice. AO4BPMN realizes the whole set of AOP concepts at the business modeling level. At this level it is already important to separate crosscutting concerns to achieve better maintainability and higher reuse. The advice code, defined in BPMN, is woven into the business process, resulting in one comprehensive BPMN process.

2.4.2 NFC Realization and Enforcement

There are different approaches in the field of NFCs and web services which are relevant for realization/enforcement. In academia and industry two prominent strategies are pursued to deal with the realization/enforcement of NFCs:

- Use Variations of the Interceptor Pattern
- Aspect-Oriented Programming (AOP)

Both strategies are mainly motivated by the challenge of dealing with the crosscutting nature of NFCs. Thus, they both provide an answer to the question how NFCs can be modularized and integrated with FCs in a transparent way, so that the code that implements FCs does not need to be changed at all.

The application of the interceptor pattern (aka Chain of Responsibility pattern [36]) is often employed in message-oriented middleware and allows for the augmentation of additional functionality in a transparent way. In web service frameworks the term handler (Apache Axis2) or filters (e.g., in the Java Servlet Specification [22]) are often used synonymously. The interceptor pattern has been adopted by many frameworks that are used in industry. The most prominent example from the Java world is the JEE Platform [83] where a user can specify interceptors in a declarative way by using annotations that can either be applied to *Session* or *Message-driven Beans*.

AOP on the other hand introduces new language constructs for the modularization of crosscutting concerns. In AOP, there are new means of modularization called *aspects*. An *aspect* realizes a concern by the combination of *pointcut* that can be defined as a query over a set of *join points* (which are well-defined points in the execution of a program) and *advice* that encapsulate the behavior added to the *join points*. Because of the crosscutting nature of NFCs, AOP is generally a well-suited approach for NFC modularization.

Baligand and Monfort [3, 44] combine the AOP and handler approach. They use a single global handler that is restrained to SOAP logic. In addition, they use a bytecode modification of the web service stubs to select appropriate aspects that enforce the requirements specified by WS-Policy. In this work, policy and aspects are used to increase web service (WS) adaptability. It is stated that existing WS technology, exemplified by handlers, is not policy-adaptable, which is a requirement when policies must be changed at runtime or the same WS must handle different policies from different clients. The answer to this problem given by Baligand and Monfort is to replace the multiple specific handlers by a single global handler whose role is restrained to SOAP logic. Weaving of aspects is applied at load time in the WS stub. A policy engine selects the appropriate aspects to enforce the requirements defined by the policy. The solution is implemented with Tomcat[4] and Apache Axis and instruments the bytecode of the WS stub through a modification of Tomcat's class loader.

2.4.2.1 Black Box Approaches

Singh et al. [85] introduce the Aspect-Oriented Web Services (AOWS) model. In their approach an *AOWebService* Requester consumes an *AOWebService* using an *AOConnector*. When the consumer needs multiple services to fulfill her request and thus multiple AOWebService providers are involved, the relevant providers are selected and bundled by an *AOComposite*. An *AOWebService* provider registers her service by depositing the *AOWSDL* interface at the *AOUDDI*. *AOWSDL* comprises a technical and a human readable descriptor specifying functional and non-functional properties of web service interfaces. It is modeled as a set of *AOComponents* each containing a set of functional and non-functional aspects. The *AOUDDI* registry locates appropriate *AOAdaptors* that are used by the *AOConnector* to serve the request with the proper protocol (for instance the Business Transaction Protocol, BTP for short). Hence, the

[4] `http://tomcat.apache.org`

AOConnector is the key role for separation of concerns. The AOWS subsystems have been modeled, analyzed, and verified in Alloy [48], which is a first-order logic based structural modeling language for expressing complex structural constraints and behaviors.

Nabor C. Mendonça et al. [59] introduce the Web Service Aspect Language (WSAL), which provides aspects that can be freely specified, implemented, deployed and executed as loosely coupled web services and thus are implementation technology-independent. Weaving is realized at network level by Web intermediary technology [46] . Supported join point types are message part, service name, service operation, service location, client location and combinations of them. Advice is implemented by WS operations and the following types are allowed: *before request*, *upon request*, *after response*, *upon response*, *after exception*, *upon exception*, and *around*. Additional context information like the intercepted message or the service location can be passed to the advice implementation operation. The context information can be used to restrict the information that is transmitted to the advice WS. Particular advice types, e.g., *before* and *after*, can be executed in parallel with the join point. Aspects are specified by an XML syntax and contain a set of join point and advice elements. The aspect weaver is implemented with IBM WBI intermediary technology.

Verheecke et al. [97] propose the Web Services Management Layer (WSML) for the dynamic integration, selection, composition, and client-side management of web services in Service-Oriented Architectures (SOA). There are different aspect types. *Service redirection aspects* specify the communication logic to swap from one web service to an alternative one or to a web service composition implementing the same logic by calling multiple services. *Service selection aspects* enforce a certain selection policy which can be of one of the types *client-initiated selection*, *non-functional service property-based selection*, *service behavior-based selection* and *service-initiated selection*. *Service management aspects* modularize concerns like encryption, billing, reliable messaging, transactions, etc., which can be enforced per service type, per composition or per web service. The aspects are implemented by JAsCo [94], an adaptive programming approach. *Combination strategies* is an imperative approach that allows for employing the full expressiveness of Java.

Henkel et al. [42] use AOP technology, namely AspectJ [54], in order to implement QoS monitoring without changing the existing service implementation. They have written a performance aspect that measures the execution time by defining two join points, one at the beginning and one at the end of a web service operation call. The advice captures the time when the request arrives at the service and when the response is returned by the service. Further, they implemented a cost aspect that counts the types of service invocations and cumulates them to calculate the monetary total cost of usage. The reliability aspect logs every invocation that ends with an exception. In conclusion, they showed exemplarily how QoS metrics can be measured when applying aspect technology to the implementation of the service.

2.4.2.2 Gray Box Approaches

Ganesan et al. [37] propose a specification language to design non-functional service concerns as distributed aspects and provide an aspect runtime using AspectWerkz [8] and SmartFrog [39]. Their distributed aspect model encompasses three components: a monitor specification, a behavior specification and an advice specification. The monitor specification contains the event name, a predicate (for example, a predicate that is true when the status of a reservation changed) and the service name. The behavior specification is a state transition system with states, transitions and mappings from states to finite set of predicates. A boolean combination of these predicates is used as hook for the advice specification which is a triple of *guard*, *local pointcut* and *advice logic*. The monitor component is realized using a dynamically woven local aspect of the composite service host. Composite services are expected to be written in BPEL.

Wohlstaedter et al. [100] introduce a new service-oriented middleware architecture for runtime web services' interoperability facilitating clients to use middleware as a service. In their architecture, which is called Cumulus, the interoperability requirements of a web service are expressed by WS-Policy (describes which interoperability protocol should be used, e.g., WS-BA) and WS-PolicyAttachment (describes at which service or which BPEL activity the requirement occurs). These requirements are enforced by middleware (mw) services that are described by metadata (for example the implemented protocols) that facilitates mw service selection. Therefore, a mw service registry matches effective policies of a client with mw service metadata. When a matching mw service has been discovered, it will be applied using the so-called mw service injection protocol (MIP) which is provided through an extra port type by each mw service. The bind operation defined by the MIP takes as an input the policy assertion that was used to discover the mw service and returns a session identifier, a specific message context and information as to whether the mw service should be added as intermediary in the incoming or outgoing message flow. A corresponding unbind operation unregisters the client from the mw service. Additionally, there are two operations to deliver incoming or outgoing messages to the mw service. If several mw services are bound, then a static order of policy assertion types is used. The architecture has been applied prototypically to BPEL (Cumulus4BPEL).

AO4BPEL [14] is an aspect-oriented extension to WS-BPEL which allows definition of aspects that modularize crosscutting concerns such as non-functional concerns. In this work, aspects are used to enforce NFCs such as security, reliable messaging and transactions. An aspect in AO4BPEL is a container for multiple pointcuts and advice constructs and can be activated at runtime. The pointcut language is XPath which allows selecting arbitrary BPEL activities. When a pointcut matches a certain join point its corresponding advice is executed. An advice is defined in BPEL and can be used to call dedicated middleware services in order to enforce NFRs. One of the unique features in AO4BPEL is that of cross-layer joint points, which span the process and service layer allowing the process and the aspect to access the SOAP messaging layer. This feature is also the key enabler for the container that integrates middleware web services (for security, reliable messaging and transaction management) into

BPEL processes. In this work the container is aspect-based and it is generated automatically from a deployment descriptor based on predefined aspect templates.

In [10] Braem et al. present Padus, an aspect-oriented extension to WS-BPEL. In Padus each activity is a potential join point; hence, it allows for structural (sequence, flow, etc.) and behavioral join points (invoke, reply, etc.). The Padus language provides a logical pointcut language to select these join points by specifying the activity type and constraints on their properties. Additionally, the pointcut language provides means to select a certain process, process instance or variable. With the advice language of Padus, additional behavior can be added to (before or after advice) or can replace (around advice) the original behavior. The advice behavior is defined in WS-BPEL and can be parameterized. It may use new variables and partner links that can be added to the process globally. Aspect composition is specified by a precedence mechanism for superimposition that is bound to an aspect instance. This allows for variance of aspects on different deployments. The approach of Braem et al. uses static weaving that enables high performance and does not require a modified WS-BPEL engine since it transforms the original BPEL document and the aspect definitions into a standard-conform BPEL process.

Erradi and Maheshwari [31] introduce AdaptiveBPEL a composition framework leveraging aspect-oriented service software composition techniques in order to enable dynamic change and configurability for composite web services. The used adaptation process is policy-driven. Aspect code is generated out of policies that are negotiated dynamically at runtime. Pointcuts are defined with XPath and advice logic is implemented as a set of web service calls.

In [11] Charfi et al. present an approach to NFCs in BPEL processes based on external policy attachment. An XPath-based selector is used to select activities that share a common non-functional requirement and a declarative WS-Policy policy is associated with this selector. The realization of the requirements is done by a SOAP middleware developed in the context of the project Colombo [23]. The supported subjects for NFCs in this work are the service subjects that are supported by WS-PolicyAttachment plus BPEL activities and partner links.

Souza et al. [88] introduced the Sec-MoSC (Security for Model-oriented Service Composition) methodology for incorporating security requirements into service compositions. The authors use *NF-Attributes* and *NF-Actions* to describe the security requirements and the design decisions, algorithms, data structures and configurations implementing these requirements. An *NF-Attribute* can be composite (in this thesis comparable to non-functional concern) or primitive. An *NF-Action* is an abstraction for the enforcement mechanism realizing the attribute. An Action also has key-value pairs to make the action configurable and thus reusable. An *NF-Statement* models constraints on *NF-Attributes,* e.g., high, medium, or low confidentiality. The authors also provide a solution for the enforcement of their model by generating first a generic WS-BPEL process with service annotations and then a platform-specific WS-BPEL and a platform-specific security configuration (which is an extension to WS-Policy). They used Apache ODE[5] and Rampart[6] as concrete execution environment.

[5] `http://ode.apache.org/`

[6] `http://axis.apache.org/axis2/java/rampart/`

2.4.2.3 General AOP Extensions

Nishizawa et al. [64] propose the concept of remote pointcuts. With this approach, it is possible to select join points on different hosts. Therefore, Nishizawa introduces a new language similar to AspectJ called DJcutter that is based on Java. The language allows for restricting pointcuts to specific hosts by introducing a new poincut designator *hosts(Host ....)* and additional reflection data about the host which the pointcut matched. DJcutter is restricted to Java and Nishizawa et al. verified it with Java RMI.

2.5 Requirements

This section collects all requirements that have been derived from related work and from the research project PREMIUM|Services. In this process, the requirements are categorized into specification and realization requirements.

2.5.1 Specification Requirements

This section will present the specification requirements.

2.5.1.1 Non-functional Requirements Specification

In typical software engineering processes, the software development lifecycle starts with the specification of requirements. Requirements are often classified either as functional or non-functional (cf. Sommerville [87]). Neither can a common consensus be reached nor is there any commonly agreed-upon definition of exactly what an NFR is, and sometimes the term goal is used synonymously (cf. Glinz [38]). This thesis supports the general definition of Franch [34] which is: Functional requirements define *what* a system should do opposed to non-functional requirements which define *how* the system should behave. NFRs can be classified by their representation, satisfaction and role [38]. The representation can be qualitative, quantitative (Mylopoulos et al. [60]), declarative or operational. The representation has an impact on how an NFR can be verified. The satisfaction can be either hard (met, not met), quantifiable or informal/leveled (see also Galster et al. [35]). The role of an NFR is either normative, assumptive or prescriptive. Approaches that allow for specifying NFRs should clarify which kind of NFR they should support. In the context of web services, a qualitative or operational, prescriptive NFR specification makes sense whereas the satisfaction depends on the respective concern (for example, service level should be quantified, security should be leveled). The need for a requirement specification has also been motivated by most of the analyzed works, e.g., GORE [93], SCA Policy Framework [5], Soeiro et al. [86], among others. In summary, it is an obvious requirement to be able to specify NFRs for web services (**Requirement S1**) because it is important for service engineers to know the exact requirements and goals. Otherwise, essential NFCs may be neglected or unnecessary NFCs could be focused on. For example, encryption should only be

used for services that require confidentiality; otherwise; unnecessary complexity and a performance reduction will be introduced. Hence, most works and also general software engineering approaches start with NFR specification. NFRs should also be associated with subjects they apply to in order to be able to determine if requirements are concerned with a service, an operation or specific message types (refer also to Requirement S3), and it should be possible to define any kind of project-specific requirement. A predefined set of NFRs is not sufficient, since usually not all NFRs can be anticipated for all kinds of services.

2.5.1.2 Non-functional Actions Specification

From the specification of requirements it is often not possible to automatically derive the concrete actions that must be taken in order to satisfy the requirements. Hence, it is crucial to define the non-functional actions that can be used in order to satisfy the previously specified requirements (**Requirement S2**). This is also applied by most state-of-the-art approaches; even many GORE approaches allow the definition of concrete actions that relate to the specified NFRs. Rosa et al. [73] define non-functional actions as software aspects such as design decisions, algorithms, and data structures, among others, or hardware characteristics concerning computer resources that affect (realize or have an effect on) non-functional attributes. This definition is constrained to algorithms and data structures that have been implemented to satisfy or contribute to non-functional requirements. NFAs should abstract the non-functional behavior that is represented by the specification in order to keep the specification as simple as possible (fostering maintainability and understandability) but should later, during runtime, be associated with a concrete software module realizing the action. This is at least required if the specification is intended to be directly used for the implementation of a system. The specification approach may offer a predefined set of available actions but should allow for any further sets of custom actions as many of the actions cannot be anticipated.

The separation of the requirements and their realization is also found in popular software development processes like the IBM Rational Unified Process [53] that differentiates the Requirements Discipline and Analysis and Design Discipline (aiming at how the requirements can be realized). The requirements definition should be framed by a requirements engineer, whereas the action definition should be framed by an expert of the non-functional domain, e.g., in case of security by a security expert. Satisfying the non-functional attributes confidentiality or response time to a certain extent is an example for non-functional requirements of a web service. Confidentiality may be realized by the NFA *Encrypt*, or the response time could be optimized by the NFA *Cache*. The particular action has an impact on the quality and level of attribute realization. For example the chosen encryption algorithm is an indicator for the quality of the confidentiality.

2.5.1.3 Web Service Subjects Specification

An important requirement for the specification of NFCs is the existence of means for defining the subjects of non-functional requirements or actions (**Requirement S3**). Without information about the subject it is not clear to what an NFA (or a non-functional requirement) applies. Different granularity levels can be distinguished for the subject specification. A subject, such as a whole service (or even a set of services), can be coarse-grained, or it can be fine-grained at the level of a service operation, its input or output or fault message, or, alternatively, a certain type of message. These subjects are all part of the WSDL interface of a web service as can be seen in Figure 2.7. Service subjects are also motivated by the WS-PolicyAttachment specification and generally by works that take the black box view on web services, for instance Ortiz and Hernandez [68].

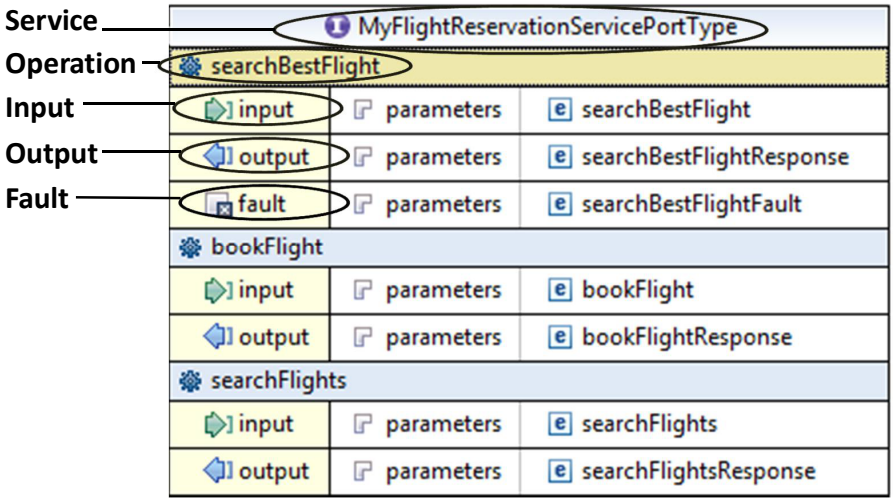

Figure 2.7: WS Subjects Identified in Eclipse WSDL Editor

2.5.1.4 NFA Interdependency Specification

Non-orthogonal NFAs are NFAs that cannot be treated completely independently from other NFAs. This means that there are dependencies between NFAs, and when they are executed in the same context, e.g., applied to the same subject, they may conflict with each other. These inconsistencies, which would occur at runtime, can already be avoided at design time by a specification approach. Hence, interdependencies between NFAs should be identified (**Requirement S4**) and specified after NFAs have been identified. In Sanen et al. [76], a set of interaction types has already been identified at an NFR level, but most of them should also be applied to NFAs. Weber et al. [98] also motivate dependencies between NFAs.

2.5.1.5 NFA Execution Order

If there is more than one non-functional requirement for a service, a set of NFAs must be applied to the same subject (c.f. Section 2.2.3, composition of superimposing NFAs/vertical composition). In this case the execution order of non-orthogonal NFAs has an impact on the result or effect the actions produce. For example consider the composition of a caching action that caches frequent web service invocations (and thus improves performance) with an accounting action that tracks all invocations per consumer in order to charge her on a per-use basis. If caching is executed before accounting, already cached invocations will not be accounted for, whereas in the reverse order all invocations no matter whether cached or not will be accounted for. As a consequence, one needs appropriate means to define the execution order of NFAs depending on the particular use case (**Requirement S5**) which is also motivated by Fox and Jürjens [33], Soeiro et al. [86] or Naveed et al. [62] (Naveed et al. refer to FCs as core concerns and NFCs as aspectual concerns). There are two kinds of execution order mechanisms that can be used. On one hand there are static execution orders implying that actions are always executed in the same order; on the other hand there are dynamic execution orders which use conditional control flow decisions based on runtime data in order to determine the execution order. The most common static order mechanisms are first-plugged-in first-executed (Courbis and Finkelstein [21]), priority-based or dependency-based order. The first-plugged-in-first-executed is the most obvious strategy for dynamic weaving approaches. The priority-based one is simple and easy to understand but limited because it is complex to add a new action between two already existing ones, e.g., if they are already using the priority number 1 and 2. With the dependency mechanism it can be defined which action precedes another action, allowing the most flexible orders.

2.5.1.6 Composite WS Subjects Specification

In the case of composite web services, there is more knowledge on the service than just its WSDL interface: The composition logic has been defined using process language standards such as WS-BPEL [2] or BPMN [67]. Taking the process definition as additional information about the service, further process-specific subjects can and should be addressed (**Requirement S6**) as can be seen in Figure 2.8.

Most identified gray box approaches motivate the requirement for process-specific subjects, but they mostly neglect service subjects. The most appropriate elements of a process are its messaging activities, which are responsible for the consumption of other web services, structured activities such as subprocesses or data stored in variables. As BPMN is the more appropriate specification language with a standardized visual syntax, it will be elaborated on the detailed requirements on the basis of BPMN. On this basis different process aspects must be regarded: the Control Flow Aspect, Data Aspect, and Functional Aspect. The Functional Aspect addresses the functionality provided by the partner services of the process.

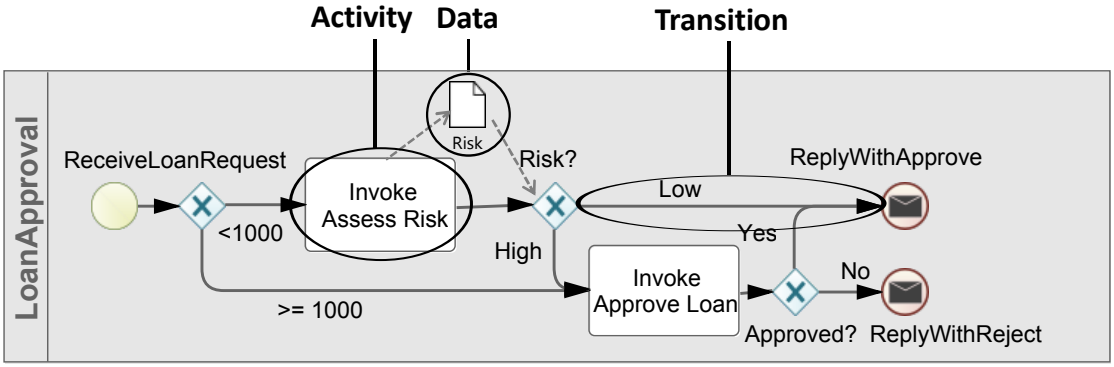

Figure 2.8: Composite Service Subjects Exemplified by the BPMN Process

Control Flow Requirements (**S6.1**): The control flow of the process defines the execution order of process components (for example activities or tasks). NFAs should be triggered at well-defined points in the control flow, for example before or after the execution of a certain flow node (node-centric view) or at a transition (for example a BPMN2 Sequence Flow) from one node to another (transition-centric view). NFAs responsible for monitoring are examples for NFAs that rely heavily on control flow. For example, there should be a flexible way to trigger a *StartTimer* NFA and the respective *StopTimer* before and after individual or sets of flow nodes (in order to capture the execution time of the node or node sets).

Data Requirements (**S6.2**): A process operates on internal or external data (variables, data objects) consumed or produced by process components. The lifetime of internal data may be bound to that of process components or instances. Most NFAs consume data by reading or modifying the data. In the black box view, only data that is carried out by messages between consumer and service is visible, but in processes data is explicitly modeled. Some NFAs are data-centric, which means that they are rather applied to a certain type of data than to a specific message exchange with a partner service. Thus, it should be possible to associate NFAs with specific data in a process. For example an Encrypt action should be associated with customer data which would imply that all messages between process and consumer or partner services containing this data would be encrypted.

Functional Requirements (**S6.3**): Process components realize atomic behavior which can be implemented by internal components (e.g., BPMN2 Script Task) or by external partner services. In case of external services the process itself acts as a service consumer and exchanges messages with the service. Hence, NFAs should be applicable at these points for incoming, outgoing or fault messages, especially if the partner service requires certain non-functional attributes such as confidentiality. Hence, it should be possible to associate NFAs with a specific messaging activity or alternatively with a certain partner (for instance when all communications with the same partner must be confidential).

2.5.1.7 NFA Control Flow Specification

Not only FCs but also NFAs follow well defined processes with control flow that should be specified explicitly (**Requirement S7**). Hence, there are two types of NFAs: composite and atomic ones. Composite NFAs define the control flow of atomic NFAs (c.f. Section 2.2.3, horizontal composition) and they do not apply to a single punctual subject, e.g., a secure conversation only makes sense for multiple operation invocations. In the black box view, only punctual subjects can be addressed with one exception: the input and its corresponding output subject. In this exceptional scenario only trivial composite NFAs like *Monitoring* can be realized, for instance, a *StartTimer* action can be applied when an incoming message arrives and a *StopTimer* action can be applied when the corresponding response is returned. Advanced composite NFAs should rather be applied to composite functional subjects such as composite web services; for example a secure conversation context (which can be seen in Figure 2.9) could be established after the instantiation of the process, all communications with partners could be secured with the *EncryptWithCtx* action, and before the process terminates, the *CancelCtx* action could finally be applied.

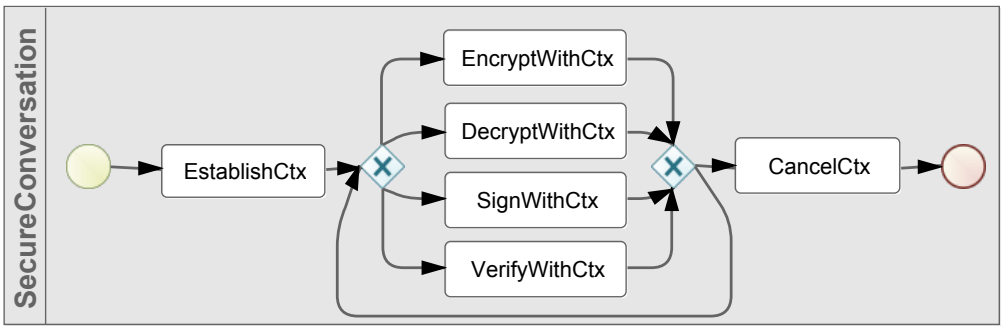

Figure 2.9: Control Flow of Composite NFAs Exemplified by Secure Conversation

2.5.1.8 NFA Data Flow Specification

Some NFAs consume and/or produce data, either from other NFAs or the functional subject. For example a *StartTimer* action could produce the captured *starttime*, a *StopTimer* action could consume *starttime* and calculate the *totaltime* and a *LogData* action could log this *totaltime* to a log file. Also data available in the functional subject may be consumable by actions; e.g., many of the mentioned NFAs need access to the message sent or received by the service. In the context of composite web services a richer set of data is available. It is for example important for particular actions to be aware of the currently executed process activity or to access particular process variables. Hence, data flow specification is necessary for a flexible composition and reusability of NFAs (Requirement S8). Especially gray box approaches, e.g., AO4BPEL, allow for accessing context data of the current execution point. The investigated works do this only on the realization level and not on the specification level. With a rich NFA specification including

the action parameters, the access to context variables should be moved to the specification level since parameter passing can be implemented on a declarative level (cf. assignments of Data Associations in BPMN2). Table 2.1 provides a summary of the specification requirements.

ID	Requirement
S1	Non-functional Requirements Specification
S2	Non-functional Actions Specification
S3	Web Service Subjects Specification
S4	NFA Interdependency Specification
S5	NFA Execution Order Specification
S6	Composite WS Subjects Specification
S6.1	Control Flow Specification
S6.2	Data Flow Specification
S6.3	Functional Specification
S7	NFA Control Flow Specification
S8	NFA Data Flow Specification

Table 2.1: Specification Requirements

2.5.2 Realization and Enforcement Requirements

During the specification of NFCs for web services it is important to capture the information that is necessary to understand which NFCs are involved and how they are composed with web services, to verify the correctness and consistency of the functional and non-functional processes and to realize the specification at runtime. For the realization new requirements arise.

2.5.2.1 Separation of Concerns

The separation of concerns (SoC [45]) principle promotes the decomposition of complex software systems into parts or modules with coherent responsibilities. Applying SoC makes complex systems maintainable, understandable, reusable etc. In general, it should be applied to both specification and realization level. However, SoC on specification level is already covered by the separation into functional and non-functional actions, requirements and so on. Applying the principle to realization level means that each concern should be implemented in its own software module whether it is non-functional or functional. However, the challenge with NFCs is that they are crosscutting and are scattered and tangled throughout the functional code. Nonetheless, they should be separated from one another and from the functional code if possible (**Requirement E1**). This principle is also followed by the aspect-oriented approaches that have been analyzed.

2.5.2.2 Weaving of FCs and NFCs

After separation of concerns into reusable, coherent modules, the separated modules must be glued together into a working application again. The mechanisms for the integration of the NFC

code into the FCs should be realized as transparently as possible (for example obliviousness [32] should be achieved; i.e., the functional code is not aware of the non-functional one) in order to avoid the tangling of functional and non-functional code (**Requirement E2**). The process of integrating NFC into FCs is often referred to as weaving and is also motivated by the aspect-oriented approaches that have been identified. There are two flavors of weaving that play an important role for web services: static weaving, which is done at design time, and dynamic weaving, which is done at runtime. On one hand, static weaving is easier to implement and performs better than dynamic weaving, but on the other hand, dynamic weaving allows the addition of NFCs during runtime without redeploying the application or the web service, which is indeed necessary for 24/7 availability.

2.5.2.3 Quantification

Another requirement is the capability to associate an NFC with FCs based on a logical condition (**Requirement E3**, also called quantification (Filman and Friedman[32]). The aspect-oriented approaches motivate this requirement. The minimal requirement concerning quantification is to be able to quantify over a set of web services and their operations. For the gray box view this is not sufficient, because quantification should also encompass internal process elements and their state. For example, quantification could state that for all processes, after executing the first task, the NFA *Encrypt* shall be executed. For expressing quantified statements, dedicated languages should be used depending on the functional concern; e.g., for web services, the XML-related XPath [19] or graphical languages for processes make sense. In the AOP domain these languages are called pointcut languages and there are different types of pointcuts: The *Call(operation)* pointcut captures the call of an operation. The execution context is the code that calls the operation. A call pointcut in the context of web services would select the point where the consumer code produces a message that is used to call the web service. The *Execution(operation)* pointcut captures the execution of an operation. The execution context is the operation that is currently executed. For web services this would correspond to the point where a message arrives at the service. The *Get(field)* pointcut selects the points in the code that read from a field. The *Set(field)* pointcut selects the points in the code that write to a field. The *Get* and *Set* pointcuts are only available for composite services which expose their variables. The *Initialization* pointcut matches the construction process of an object. Web services do not expose any state information in the black box view, but in the gray box view, instances of processes become visible and could be matched. The *Handler* pointcut matches the execution of exception handlers. The corresponding web services concept is faults which could be matched with this kind of pointcut designator. There are also other primitive pointcut designators, but they can only be applied to web services regarded from white box view, for example *cflow* or *staticinitialization*. All pointcuts that were mentioned here were taken from AspectJ [54]. In other aspect-oriented approaches similar constructs exist.

2.5.2.4 Superimposition

In the previous section it was already motivated that multiple NFAs could apply to the same subject and that in this case the execution order should be specified. The superimposition (Nagy et al. [61], Bergmans and Aksit [7]) of NFAs should also be supported at runtime (**Requirement E4**). This requirement has also been described and motivated by a subset of the approaches which have been analyzed in related work, e.g., AO4BPEL.

2.5.2.5 Transparent Weaving with Distributed Web Services

Another group of requirements stems from the distribution and platform-independence characteristics of web services. According to Papazoglou [69] web services are distributed, loosely coupled to the consumer and technology-neutral. Hence, one requirement for the realization of NFCs for web services is that the solution be able to cope with the distribution; e.g., it should be possible to reuse the implementation of an NFC for multiple web services even if they are running on different machines (**Requirement E5**).

2.5.2.6 Programming Language Independence

Another requirement is that the components that implement the NFCs should be platform-independent and thus can be applied to the web services independently of the programming language that is used for their implementation (**Requirement E6**). Table 2.2 provides a summary of all enforcement requirements.

ID	Requirement
E1	Separation of Concerns [52] (Modularity [72])
E2	Weaving of FCs and NFCs
E3	Quantification [32]
E4	Superimposition (Composability [72])
E5	Transparent Weaving with Distributed WS
E6	Independence of Programming Language

Table 2.2: Enforcement Requirements

2.5.3 General Requirements

As already motivated in Section 2.3 an abstract model should be transformed into executable code. This can be done either manually or in an automated way. Generating code out of the model is an automated way which helps to keep source code and model in sync. This allows the evolution of the model and the code. Manual transformation is rather error-prone and requires great effort. Thus, code generation is a general requirement which must be supported. Alternatively, a model can directly be interpreted when it contains enough information to be

executable. The problem with this approach is that in the case of directly executable models the model becomes quite complex and is not abstract enough.

2.6 Evaluation of Related Work

In this section, the work presented in Section 2.4 is evaluated against the requirements identified in Section 2.5.

2.6.1 NFC Specification

2.6.1.1 Specification for Web Services (Black Box View)

In general, GORE approaches support only Requirement 1, but some of them (e.g., the NFR framework through satisficing goals) also support actions specification (Requirement 2) or interdependencies between requirements (Requirement 4 partly supported). However, the experience is that it makes more sense to handle interdependencies later when concrete actions have been defined, because these actions reveal more useful information with respect to realization than abstract requirements. In approaches that focus on realization, particular soft goals like usability do not make sense since no actions can be deduced to improve the usability of web services.

The WS-Policy framework realizes Requirements S2 and S3. It allows for the specification of NFAs in terms of assertions and facilitates different functional subjects as defined by WS-PolicyAttachment. Requirement S1 is not in the scope of WS-Policy as this language focuses on NFA definition and not on the definition of non-functional requirements, concerns or attributes. Interdependencies are specified in the concrete domain specific policies, but there is no general interdependency concept with different types in the WS-Policy Framework specification. For example, particular actions in WS-ReliableMessaging [26] require the use of WS-Addressing [9]. This is not specified formally but only in a textual form (S4 not supported). WS-Policy cannot cope with S5, because the order of actions in a policy can only be expressed by the semantics of its declarative assertions. An example for this can be found in the WS-SecurityPolicy in which two assertions *EncryptBeforeSigning* and *SignBeforeEncrypting* are defined to reflect the order of the non-functional actions *Encrypt* and *Sign*. Thus, for each possible reasonable combination of NFCs, a new assertion must be invented. A similar problem occurs when regarding the interaction between different concerns. For advanced security requirements, it is further necessary to secure the reliable messaging mechanism which is also solved by additional assertions. In order to combine security with reliable messaging, the WS-ReliableMessaging Policy [27] defines a *SequenceTransportSecurity* assertion which indicates that also the messages produced by the reliable messaging protocol should be secured. Another drawback with this approach is that the WS-ReliableMessaging Policy contains security related assertions breaking the separation of concerns principle. WS-Policy is a specification that has been designed for web services

as black box and not for composite BPEL processes [14]; hence, it cannot cope with Requirement S6. Requirements S7-S8 are not taken into consideration and are left to implementations supporting WS-Policy.

SCA makes use of WS-Policy and thus supports the same requirements. In addition, it provides a simple mechanism to define non-functional requirements and hence also supports Requirement S1. An *intent* and the policies can be attached either to a service or to its operations, meaning that they apply to the communication between services. Alternatively, a policy can also be attached to the services implementation.

ProcessNFL [73] satisfies Requirements S1 and S2 because non-functional attributes and actions are first-class entities of the language. S3 is not supported because neither web service specific subjects nor any other functional subjects can be defined for non-functional actions. S5 is also not supported since composition takes place on attribute level and not on action level. Consequently, Requirements S4 and S6-S8 are not supported either.

Sanen et al. [76] support the specification of NFRs (Requirement S1 supported), e.g., Authentication, Authorization, Confidentiality etc. They do not introduce the concept of NFAs (Requirements S2, S5, S7-8 not satisfied) but rather directly the concrete components that realize the requirements. The approach is not applied to web services; thus, no web service subjects are supported (Requirements S3 and S6 are not supported). The interdependency specification is done at a requirements level rather than on the action level which partly satisfies Requirement S4.

Soeiro et al. [86] satisfy Requirement S1 because they focus on the specification of concerns which are defined on the requirements level. How a particular concern is realized is not taken into account (Requirement S2 not satisfied). On the other hand, Soeiro et al. define concern interdependencies such as Concern A requires Concern B and contributions (positive and negative) between concerns which can lead to conflicts at match-points where two concerns coexist that contribute negatively to each other (Requirement S4 supported). Furthermore, composition rules such as sequential, parallel or interrupted orders of NFCs can be specified (Requirement S5 satisfied). Subjects (called match-points) for the NFC are arbitrary FCs. This work focuses on aspectual concerns in general; hence, no subjects specific to web services are taken into consideration (Requirements S3, S6-S8 not addressed).

Fox and Jürjens [33] use composition filters at the modeling level in order to specify the weaving order of a security concern to its base concern. Generally, their work can also be used for other concerns (hence Requirement S2 is supported) and supports execution order specification, but only in a limited way: Only static, sequential orders are mentioned (Requirement S5).

With UML, Requirements S1 and S2 can be satisfied. Requirements can be specified in UML for example by use of stereotypes and profiles (cf. Cysneiros et al. [24]). For all UML-profiles-based approaches, it is generally possible to apply the stereotypes and their tagged values to arbitrary UML meta classes, e.g., *Class* or *Operation* representing a web service and its operations which satisfies S3. Since UML2 it is possible to apply multiple stereotypes to one

class, but the order of stereotypes cannot be expressed (S5). The interdependencies could be expressed through the dependency relationships between classes, but most of the predefined types are not adequate for defining NFA dependencies (S4 partly satisfied). If the composition logic of composite services had been described using UML2 Activity Diagrams (or State Diagrams), the individual actions could have been annotated with stereotypes representing non-functional actions. In this case control and data flow could also theoretically be addressed. Since this is only a theoretical assumption, S6-S8 is rated as partly satisfied.

Ortiz and Hernandez [68] support Requirement S2 and S3. The non-functional requirements and attributes are not described in their work (S1 not supported). They use the term extra-functional properties which corresponds more to the term of NFAs rather than to non-functional requirements or attributes. The extra-functional properties can only be ordered statically by their priority (S5 partly supported). Composite web services are not addressed by this work (S6-S8 not fulfilled).

In Table 2.3 an overview of the works and the grade (+ supported, (+) partly supported, − not supported) of the specification requirements fulfillment is given. The approaches by Naveed et al. [62], Sanchez et al. [75], Bergmans and Aksit [6] are not evaluated in terms of requirements because they represent only very general and abstract concepts.

	GORE	WS-Policy [95]	SCA [5]	Rosa [73]	Sanen [76]	Soeiro [86]	Fox [33]	UML [66]	Ortiz & Hernandez [68]
S1	+	−	+	+	+	+	−	+	−
S2	+	+	+	+	−	−	+	+	+
S3	−	+	+	−	−	−	−	+	+
S4	(+)	−	−	−	(+)	+	−	(+)	−
S5	−	−	−	−	−	+	(+)	−	(+)
S6	−	−	−	−	−	−	−	(+)	−
S7	−	−	−	−	−	−	−	(+)	−
S8	−	−	−	−	−	−	−	(+)	−

Table 2.3: Evaluation of Work on NFC Specification (Black Box View)

2.6.1.2 Specification for Composite Web Services (Gray Box View)

The work presented by Chollet and Lalanda [16] supports requirements S1 and S2 by defining their own concern metamodels (security, logging, monitoring). These concern metamodels define classes for requirements and actions that are related via model references. The NFR classes are then added to the composition model by the use of annotations. Because NFAs are referenced by the corresponding NFRs, one can choose alternative NFAs to be mapped to annotated activities (Requirement S6 supported). Because this approach is requirement-centric, it does not support specifying if an action should be activated before or after the execution of an activity. For example, an authentication requirement can be specified that is realized by username and

password or signatures and an email sending task could be annotated with the authentication requirement. Service subjects cannot be specified because only process tasks are valid targets for annotations (Requirement S3 not supported). Requirement S4 has not been addressed because one cannot model the execution order between actions that enforce a requirement specified by an annotation. Also S7-S8 is not supported because with means of simple annotations only punctual requirements can be expressed (and not stateful ones).

Weber et al. [98] focus on composite web services described in BPMN; hence, web service subjects are not supported (Requirements S3). Requirement S1 and S2 are also not supported: NFRs are not regarded and atomic NFAs are not considered. However, Weber et al. focus on composite NFAs described with state diagrams (Requirement S7 supported) and also dependencies can be described between composite NFAs (Requirement S4 weakly satisfied). BPMN tasks can be annotated with the states from the state diagram which supports Requirement S6, but there is no explicit order between these annotations (Requirement S5 not satisfied). Due to the use of state diagrams there are no means to define data flow (Requirement S8).

AO4BPMN [13] (and also the approach of Weber et al.) addresses crosscutting concerns that require a change in the process logic (e.g., adding new tasks to the core process) and are not necessarily non-functional. AO4BPMN does not support the specification of requirements (S1 not supported). However, non-functional actions can be specified as tasks used in an advice specification (S2 supported). An advice cannot be applied to web service subjects (S3 not supported) but can be woven with the BPMN process elements (S6 supported). Interdependencies and ordering of advice elements is not supported (S4 and S5). Control flow and data flow of composite actions is also not supported (S7 and S8).

With the AO4BPEL deployment descriptor [14] requirements can be specified and in addition the concrete actions (corresponding to NFAs) can be selected; hence, Requirement S1 and S2 are fulfilled. The selector does not allow specification of web service subjects for the selected requirements (S3 not satisfied). It is possible to customize the order of NFAs using a priority-based mechanism which is static and sequential. However, there are restrictions with respect to the type of requirement; e.g., certain generated aspects cannot be influenced. One reason is that they must be the first or last one (for example reliable messaging) in the sequence, or that one requirement in the deployment descriptor could generate several aspects, with possibly different priorities (S5 partly supported). Interdependencies cannot be described, and thus S4 is also not supported. However, S6 is supported because selectors can select arbitrary process activities with an XPath expression. Neither NFA control flow nor data flow can be specified, because NFAs are black boxes for AO4BPEL (Requirements S7 and S8 not supported). Although transactions and secure conversations are supported (which are indeed composite), the atomic actions cannot be accessed and associated with subjects. Instead, these actions are always bound to composite structures in BPEL processes (scopes or sequences). For example, a transaction can be started and committed while a scope activity is executing, and all nested activities will automatically participate in this transaction.

The non-intrusive Policy Attachment for BPEL [11] uses WS-Policy for specifying NFAs. These policies can either be attached to web service subjects or to BPEL processes. Hence, the Policy Attachment for BPEL evaluation results are the same as those for WS-Policy (S2 and S3 satisfied) plus additional support for S6.

Sec-MoSC [88] uses composite *NF-Attributes* to describe the requirements of processes (S1). *NF-Attributes* are realized by *NF-Actions* (S2). Web service subjects are not supported because Sec-MoSC focuses explicitly on business processes (S3 not satisfied, S6 supported). Interdependencies/constraints on the composition of multiple *NF-Actions* are not supported, and nothing is said about the execution order of *NF-Actions*. *NF-Actions* are atomic and thus no control flow nor data flow can be described (S7 and S8 are not supported).

In Table 2.4 an overview of the works and the respective degree of fulfillment for the specification requirements is presented (+ supported, (+) partly supported, − not supported).

	Chollet [16]	Weber [98]	AO4BPMN [13]	AO4BPEL [14]	Charfi07 [11]	Sec-MoSC [88]
S1	+	−	−	+	−	+
S2	+	−	+	+	+	+
S3	−	−	−	−	+	−
S4	−	(+)	−	−	−	−
S5	−	−	−	(+)	−	−
S6	+	+	+	+	+	+
S7	−	+	−	−	−	−
S8	−	−	−	−	−	−

Table 2.4: Evaluation of Work on NFC Specification (Gray Box View)

2.6.2 NFC Realization

2.6.2.1 NFC Realization (Black Box View)

An aspect can be used to encapsulate crosscutting behavior and thus is a perfect match to provide programs with a clear separation of functional and non-functional concerns (Requirement E1). The integration of aspect behavior into the functional behavior can be done without changing the functional code by means of pointcuts (Requirement E2). Pointcuts can be understood as a facility to query over join points. Join points, on the other hand, are well-defined points during the execution of a program (e.g., an execution of an operation). In a single query, multiple join points can be selected (quantification, Requirement E3) and multiple aspects can point to the same join point (superimposition, Requirement E4). Requirement E5 and E6 cannot be addressed by AOP in general, as they rather depend on the concrete implementation.

The approach of Nishizawa et al. [64] supports E1-E4 only partly because their approach is AOP-based but has not yet been applied to web services. In summary, DJcutter supports Requirement E5, but E6 still remains unsatisfied since DJcutter is based on Java and hence

is not applicable to arbitrary programming platforms; e.g., web services implemented in other programming languages cannot use DJcutter.

Singh et al. [85] provide a heavy-weight, aspect-oriented extension to the well-known web service roles and specifications. Therefore, no specific programming language must be assumed (E6 satisfied and E1-E4 supported through use of AOP). Distribution is not supported by their weaving strategy (E5 not supported).

Mendonca et al. [59] provide a platform-independent (Requirement E6) solution that also supports distributed join points through the *service location* type (Requirement E5), but they do not consider composite web services. Since their approach is based on AOP for web services E1-E4 are also supported.

The aspects in the approach of Verheecke et al. [97] are implemented by JAsCo [94], an adaptive programming approach which supports Superimposition (E4) by *Combination Strategies*. *Combination Strategies* is an imperative approach that allows for employing the full expressiveness of Java. E1-E3 are also supported by their aspect-oriented approach.

In the approach of Henkel et al. AspectJ is used to implement non-functional concerns (E1-E4 supported). Weaving of aspects into distributed join points is not supported, because AspectJ does not support this (E5 not supported). Requirement E6 is also not supported, because the approach directly depends on Java due to the use of AspectJ.

The work of Baligand et al. [3] and Hmida et al. [44] integrate declarative policies by using handler technology. With this approach, it is not necessary to change the implementation of web services, thus adaptability at runtime is facilitated. The handlers are written in Java and are integrated tightly with the SOAP framework, making their solution technology dependent. The solution cannot be reused for different technology setups (e.g., Java, .NET), and hence Requirement E6 is not satisfied. The handler solution does not support distribution either (E5 not supported).

In Table 2.5 an overview of the works and the grade (+ supported, (+) partly supported, − not supported) of the realization requirements fulfillment is given.

	Nishizawa [64]	Singh [85]	Mendonca [59]	Verheecke [97]	Henkel [42]	Baligand [3]
E1	(+)	+	+	+	+	+
E2	(+)	+	+	+	+	+
E3	(+)	+	+	+	+	+
E4	(+)	+	+	+	+	+
E5	+	−	+	−	−	−
E6	−	+	+	−	−	−

Table 2.5: Evaluation of NFC Realization Work (Black Box View)

2.6.2.2 NFC Realization (Gray Box View)

In Ganesan et al. [37], there is no Programming Language Independence (E6) because SmartFrog [39], a Java-based framework, is used for the implementation and thus only Java-based web services can be woven with aspects. The superimposition of several aspects is not discussed in their work and thus Requirement E4 is not supported. However, there is a distributed aspect model (DAM) which enables distributed join points (E5). Since this approach is based on aspects, E1-E3 are well covered.

In the implementation of Wohlstadter et al. [100], the BPEL engine must interpret additional local handler wrappers for activities (for instance, the scope activity), and hence this work partly supports Requirement E6. Distributed join points (Requirement E5) are not supported. E1 is supported because NFCs have been modularized into middleware services. E2 is not supported because in the process definition additional handlers must be specified (transparency not supported), and neither quantification nor superimposition are supported (E3 and E4 not supported).

AO4BPEL [14] strictly follows the separation of concerns principle because the code for the realization of NFCs is modularized into middleware services (Requirement E1). The integration of FCs (the BPEL process) and NFCs is achieved by pointcuts (Requirement E2), and the code is modularized by aspects. Quantification is supported by the pointcut language which also allows quantifying over more than one process (Requirement E3). Superimposition is supported by a static ordering mechanism similar to the one provided in AspectJ, so that each concern has a predefined priority. It is not always possible to change the order for each aspect individually, especially when multiple aspects are generated out of one requirement in the deployment descriptor (Requirement E4 partly solved). Requirement E6 is partly solved. On one hand, the middleware services are implemented as web services and thus are per se independent of the programming language; on the other hand, it is presumed that the web service exposes its composition logic as BPEL. Hence, AO4BPEL is not able to enforce NFCs for black box web services. The integration of aspects with distributed web services is not possible in AO4BPEL (E5 not supported).

In Padus (Braem et al. [10]), advice code is directly woven into the BPEL document. This enables, on one hand, separation of concerns on the specification level, but on the other hand changes the original BPEL source document and intermingles the functional and non-functional concerns. Hence, the BPEL process running on the BPEL engine is not purely functional (E1 partly supported). Padus supports static weaving (E2 supported) and quantification via its pointcut language (E3 supported). Padus defines a static precedence mechanism similar to AspectJ (E4 partly supported). Distributed join points are not supported (E5). A main benefit of this approach is that an unmodified, standard BPEL engine can be used. Nonetheless, this approach is completely dependent on web services implemented in BPEL and does not support other languages (E6 partly satisfied).

In Adaptive BPEL by Erradi and Maheshwari [31], an augmented BPEL engine is used to implement runtime weaving of aspects (E2 fulfilled). The pointcut language is XPath (E3

supported). Aspect advice is a simple web service call. The aspects are generated out of WS-Policies specifying the non-functional capabilities required at a particular execution point of the BPEL process. Hence, separation of concerns (E1) is sufficiently supported. Superimposition and weaving with distributed services is not supported (E4 and E5 not supported). The platform independence is, as in AO4BPEL, only partly supported because the implementation depends on an instrumented BPEL engine which supports aspects (E6 partly supported).

The non-intrusive Policy Attachment for BPEL [11] strictly separates NFCs from the BPEL process since NFCs are realized by policy handlers (E1 satisfied). The weaving is completely transparent. The BPEL process engine merely fires events when process activities are executed, and the policy handlers listen to them (E2 fulfilled). Quantification is also supported by using XPath to specify attachment targets (E3 supported). Superimposition is only supported by means of static orders (weak support for E4). E5 is not supported; one can quantify over a set of processes, but they must run on the same BPEL engine. Programming Language Independence (E6) is only weakly supported because an instrumented BPEL engine is required. In Table 2.6 an overview of the works and the grade (+ supported, (+) partly supported, − not supported) of the realization requirements fulfillment is given.

	Ganesan [37]	Wohlstadter [100]	AO4BPEL [14]	Padus [10]	AdaptiveBPEL [31]	Charfi07 [11]
E1	+	+	+	(+)	+	+
E2	+	−	+	+	+	+
E3	+	−	+	+	+	+
E4	−	−	(+)	(+)	−	(+)
E5	+	−	−	−	−	−
E6	−	(+)	(+)	(+)	(+)	(+)

Table 2.6: Evaluation of Work on NFC Realization (Gray Box View)

2.6.3 General Requirements

The requirement of code generation is supported by only a few of the selected related works. AO4BPEL generates aspect code (in WS-BPEL) out of an XML-based deployment descriptor. Sec-MoSC generates WS-Policies out of an extended BPMN model to configure the underlying Axis2 SOAP framework. Ortiz and Hernandez generate AspectJ aspects, WS-Policies or handler configurations out of a UML class diagram annotated with stereotypes. In AdaptiveBPEL, aspects are generated out of WS-Policies and Padus generates BPEL and aspects into BPEL.

2.7 Conclusion

In this chapter, the problem statement for the research topic being addressed by this thesis has been presented. The research topic, which deals with the composition of non-functional con-

cerns in web services has various dimensions. The most important dimensions that have been identified in this chapter are:

- **Contrasting views on web services** The view (black, gray or white box) on web services has an influence on the NFCs which can and should be addressed.
- **Specification vs. realization/enforcement** Abstraction in a model showing only relevant concepts vs. executability of the model to enforce the modeled properties at runtime.
- **Different types of composition** There are different types of composition: *composition of functional concerns with NFAs*, *composition of superimposing NFAs* and *composition of composite NFAs*. Especially, the *composition of superimposing NFAs* is not supported by most of related work or at least very basic. The *composition of composite NFAs* is not supported at all.

Furthermore, related work has been investigated in order to find weaknesses in the state of the art. The most important (partly) unresolved problems can be summarized as follows:

- **Fine-grained composition of NFAs** Implementations of pure industrial standards often hide the composition logic and implementation of NFAs. This limits their composability and reusability.
- **Dynamic control flow between NFAs** Superimposing actions must be executed in a specific order or based on particular conditions. This control flow must be explicitly specified.
- **Interdependencies between NFAs** The composability of NFAs may be restricted by constraints. Violations of these constraints must be discovered by design time.
- **Platform independence** Many solutions assume too much information about web services; e.g., they require the particular programming language being used for the implementation of the web service.

In this chapter further requirements have been found which are supported by a subset of the investigated approaches. However, most often there is not a single approach which is capable of supporting all or at least most of the requirements. On one hand, different approaches could be combined, but on the other hand, this may lead to insufficient integration of heterogeneous solutions. This fact motivates the invention of a novel approach for the composition of non-functional concerns in web services.

In the next chapter, the solution for the open research problems described in this chapter is presented. This solution supports the different dimensions and strives to solve the open research problems which have been identified.

Chapter 3

NFComp: Modeling Executable Non-Functional Action Compositions

3.1 Introduction

In this chapter, a novel approach for the composition of non-functional concerns is presented. This approach, called NFComp for the sake of brevity, addresses the shortcomings and problems identified in the last chapter. NFComp focuses on web services as target components for composition with non-functional concerns.

This chapter is structured as follows: First, the general aspects of the approach are presented. These parts are not specific to web services but can be applied to all kinds of component-based software applications. Then, the web-service-specific aspects of NFComp are described in detail for each view: black, gray and white box. The white box view, however, is not only applicable for web services (written in Java), but generally for component-based software written in Java. Thus, the white box view shows how to extend NFComp in order to support more than just web services. In the following, the specification and realization as well as the different types of composition are taken into account. Parts of this chapter have been published in [80, 81, 78].

3.2 NFC Composition in Component-based Software Applications

NFComp defines a process model for the specification and realization of non-functional concerns. This process model is divided into several phases involving different user roles. Each phase has a well-defined input and output. There are four specification and two realization phases (depicted in Figure 3.1). The process starts with the *Requirements Specification* phase in which the *Requirements Engineer* specifies non-functional requirements in form of informally described non-functional attributes grouped by NFCs. In the second phase *Action Definition*, the *Non-functional Domain Experts* (e.g., Security, Performance, Reliability experts) identify and specify how, i.e., by which action the requirements can be realized. In the *Action Com-*

position phase, the *Application Provider* (to a certain extent in collaboration with the *Domain Experts*) composes the actions according to the concrete requirements and use cases. Composing — in this phase — means to define the order and control flow of the actions when executed together at the same functional execution point. During the fourth phase, *Action To Application Mapping*, the *Application Provider* takes the actions and maps them to an application or a specific component of this application. In the *Action Realization Mapping* phase, the *Domain Experts* bind the actions to concrete software components that implement the functionality for the respective action. Finally, in the *Generation of NFC Realization Code* phase, the *Application Provider* generates code that enhances the target components with the specified NFAs and activities. The prerequisite for the last phase is that the *Action to Service Mapping* and *Action Realization Mapping* phases have been completed successfully. In the following, the distinct phases are described in more detail.

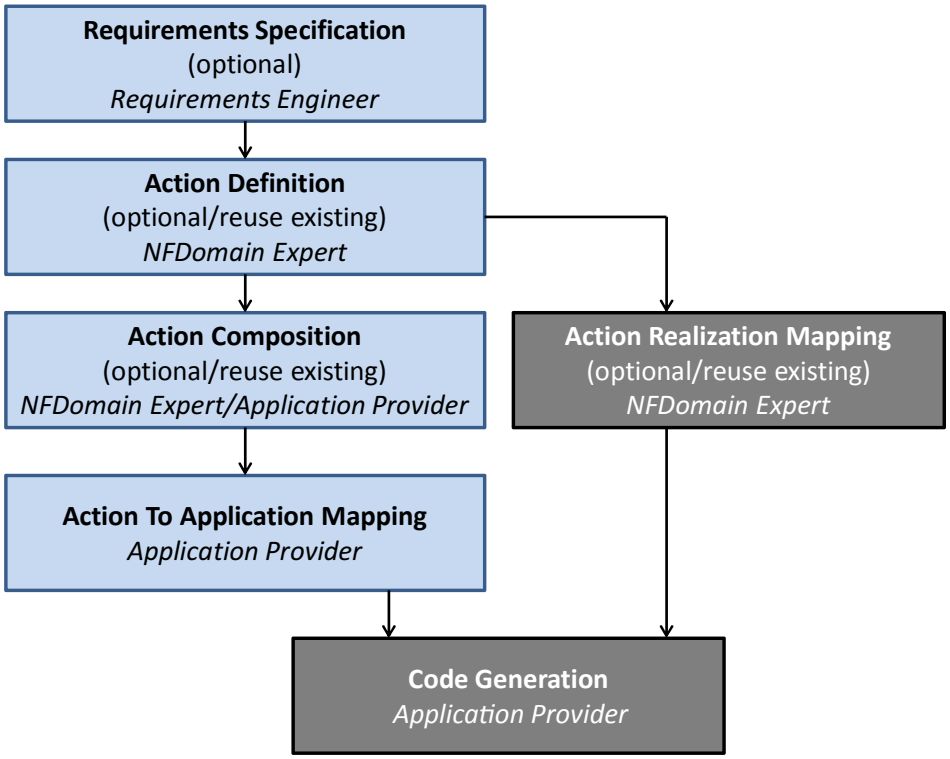

Figure 3.1: Approach in a Nutshell: Phases and Involved Roles

3.2.1 Requirements Specification

In most common software engineering processes such as the Rational Unified Process [53], there is a dedicated role called *Requirements Engineer* which is responsible for the analysis of the requirements for the developed software components. These system requirements are systematically collected in a requirements document. In general, there are two types of requirements: functional and non-functional ones. For the *Requirements Specification Phase*, only the

non-functional ones are of interest. The expert takes these non-functional requirements from this document and defines the requirements in terms of non-functional attributes. A non-functional attribute is defined by an informal text which briefly describes the requirement. However, any kind of formal expression could be used instead. An attribute is always part of a particular concern. This concern is rather coarse-grained and abstract and described by its non-functional attributes. For each identified concern it is presumed that there is a *Non-Functional Domain Expert* that has deep knowledge in this area.

Input and output artifacts: The input for the *Requirements Specification Phase* is the requirements documentation containing the non-functional system requirements. The output is a requirements model which is an instance of the requirements metamodel shown in Figure 3.2. This model is defined as Ecore Diagram, already introduced in Section 2.2.4. The requirements metamodel defines three elements: The *Non-functional Requirements Model* is the root element containing multiple *Non-Functional Concern* elements. Each *Non-Functional Concern* contains a set of *Non-Functional Attribute* elements. Concerns and attributes define a name attribute of the type *EString*. Attributes additionally define the description attribute which can be used to include a more detailed explanation of the attribute.

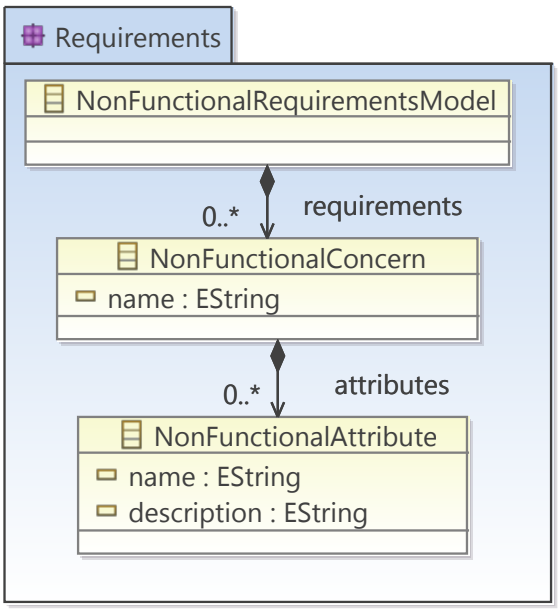

Figure 3.2: The Requirements Metamodel

Requirements Notation: Requirements are specified in terms of concerns and attributes. A concern is represented by a rounded rectangle. A concern can define multiple attributes which are represented by rectangles with two compartments: One for the name of the attribute and one for the description. The notation elements are depicted in Figure 3.3.

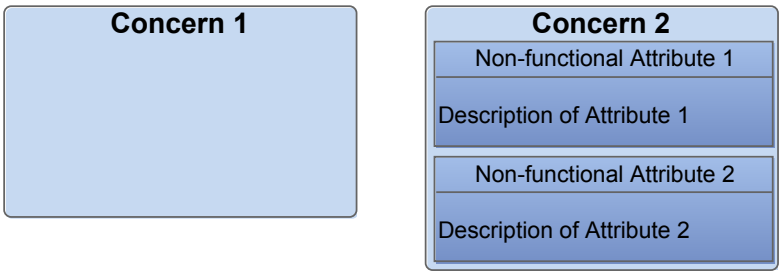

Figure 3.3: Requirements Notation

Example: A *Requirements Engineer* specifies two concerns, *Performance* and *Security*. Then she assigns the relevant concrete attributes to each concern. She adds the *Performance* attribute *LowResponseTime* and the *Security* attributes *Integrity* and *Confidentiality*.

3.2.2 Action Definition

After the specification of the requirements in terms of concerns and attributes the *Non-functional Domain Experts* define the contributing non-functional actions that affect or satisfy the attributes for each particular concern.

3.2.2.1 Action Properties

Each NFA has properties that capture additional knowledge on the nature of each particular action with the purpose of supporting the *Application Provider* later when she defines valid action compositions. The general properties of NFAs are explained in the following.

Name: The unique name of the action.

AffectedAttributes: The associationType property defines which non-functional attributes are affected by the NFA and how they are affected. There are four ways an NFA may affect an attribute (inspired by the NFR framework [18]):

- It may satisfy the attribute completely. For example, the *Authorize* action satisfies the *AccessControl* attribute completely.
- It may contribute positively to the attribute. For example, *Caching* contributes positively to the *LowResponseTime* attribute since it improves it. However, there may be other actions that can also contribute to the same attribute, e.g., *LoadBalancing*.
- It may contribute negatively to the attribute. For example, *Encrypt* contributes negatively to the *LowResponseTime* attribute because it will increase the response time.
- It may deny the attribute completely. For example logging may deny the *Privacy* attribute.

The affected attribute property helps to predict the quality properties of the components enhanced by an action and connects the action model with the requirements model.

Direction: The direction property restricts the mapping of the NFA to a certain direction with respect to the functional behavior. The motivation for this property is that there are actions that only make sense for a specific direction. For example, it does not make sense to map *Encrypt* to a functional behavior which consumes an incoming message from an external or remote component. Another example would be an action *VerifySignature*, which does not make sense for outgoing messages but only for incoming ones. The set of direction types is explained in the following. *None* defines that there is no restriction with respect to the direction. *Before* and *After* define that the action must be mapped before, respectively after, a functional concern. *Fault* prescribes that an action can only be executed when the execution of functional behavior results in a fault/exception or error. *In*, *Out*, *In_Out* are messaging directions which restrict the mapping of an action to ingoing messages, outgoing messages or both.

Impact: An action may have different kinds of impacts on the messaging process or the message itself. The impact can be data-related or control-flow-related. Possible values for data-related impacts are *Read*, *Add*, *Remove* and *Modify*. An example for control-flow-related ones is *Block*. The general *None* impact indicates that an action does not have any data or control flow-related impact (for example an action that simply counts invocations of external components). A *Read* action reads the contents of the message but does not change it (for example a *Log* action). An *Add* action adds parts to the messages such as a signature (for example a *Sign* action). A *Remove* action removes parts from the message header (for example after verifying the signature the action could remove the signature from the message) or body (a *ApplySecurityFilter* action removes parts of the response message because of security restrictions). *Modify* changes parts of the message (*Encrypt* makes the message unreadable). Actions that *Block* may decide not to deliver the message to the service itself but instead respond with a fault message (such as an *Authorize* action). The main purpose of the impact property is to determine possible composition conflicts when combining actions. For example, if only actions with impact *Read* and *None* are involved, no composition conflicts will occur, but if *Read* and *Modify* actions are combined, a modify action has an impact on all *Read* actions that are applied thereafter.

Input and output artifacts: The input for the *Action Definition Phase* is a requirements model conforming to the metamodel shown in Figure 3.2. The output is the action model which is an instance of the metamodel shown in Figure 3.4. This metamodel formalizes the concepts that have been presented so far. The root element *NonFunctionalActioModel* contains *NonFunctionalBehavior* elements. The *AttributeAssociation* represents the connection between *NonFunctionalAttribute* elements and behaviors. The association additionally defines the *associationType* attribute. The abstract concept of *NonFunctionalBehavior* represents all

kinds of non-functional behaviors such as actions. It has properties name, direction and impact inherited by the *NonFunctionalAction* concept.

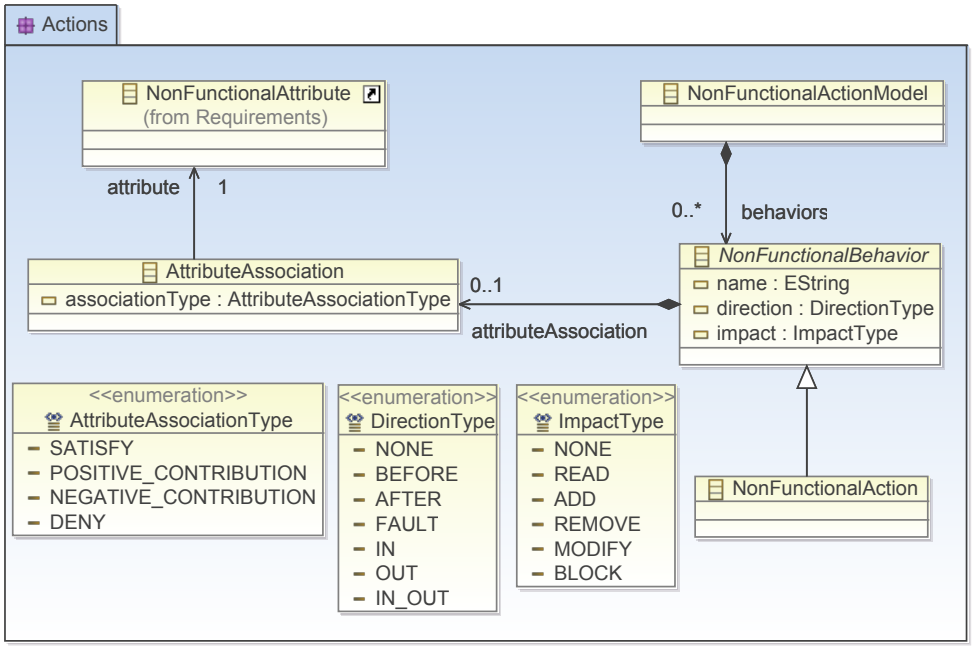

Figure 3.4: Metamodel of Actions and Properties

3.2.2.2 Action Interdependencies

After each expert has defined her actions the domain experts must determine possible interdependencies between the defined actions. Two types of such interdependencies should be distinguished: single concern and cross concern interdependencies. The former is defined between actions from the same NFC (for instance, two security actions) and the latter between actions belonging to different NFCs (for example, between *ReadFromCache* and *Encrypt* actions). It is generally more difficult to find cross concern interdependencies, because different non-functional domain experts need to collaborate in this case. The types of interdependencies (based on Sanen et al. [76] and applied to actions instead of requirements) are listed and explained in the following.

Choice: When two actions are related to each other with the *choice* interdependency type, the semantics is that both actions are basically doing the same with respect to the non-functional attributes they realize. Hence, two actions connected with *choice* should not be chosen for the same functional subject, but if so would not cause serious problems. An example would be a *StrongEncrypt* action and a *WeakEncrypt* action that would both fulfill the *Confidentiality* attribute. Combined, they would cause a message to be encrypted twice, which would not increase the level of confidentiality significantly.

Composition Implications: When an NFA is chosen for a functional subject and a modeler wants to add another NFA that is of type choice then she is warned that there is already another action with a similar effect.

Conflict: Two actions may conflict when one affects a certain non-functional attribute in a positive way (for example *ReadFromCache* and *ResponseTime*) and the other affects the same attribute in a negative way (e.g., *Encrypt* and *ResponseTime*).
Composition Implications: When two actions related to the *conflict* interdependency are combined, the modeler is warned that these actions conflict. Based on this warning she can decide to remove the conflicting action depending on the requirements. For instance, when the requirement for an excellent response time outweighs the requirement for confidentiality, she may decide to remove the *Encrypt* action.

Mutex: The use of one action may exclude the other. If one of the two actions that excludes the other is added to a functional subject, the usage of the excluded action must be prohibited. The difference to choice is that the actions must not be used for the same subject, whereas in choice, it is not reasonable but possible. An example for mutually exclusive actions is the usage of two different *Accounting* actions. If both are used for the same subject, the consumer's invocations would get billed twice.
Composition Implications: If an NFA is chosen for a functional subject, then all corresponding NFAs that are interdependent with type *mutex* must not be added to the subject.

Assistance: An action may assist another action if it has a positive impact on the same non-functional attribute. An example would be a *Caching* action and a *LoadBalancing* action that both influence the response time in a positive way.
Composition Implications: The advantage of knowing that two actions assist each other is that whenever the modeler makes use of an action the modeling tool may propose the use of one of its assisting actions to further improve/satisfy the non-functional attribute that is realized by the actions.

Requires: The *requires* interdependency between two actions defines that the use of one of the actions requires the use of the other. An example is an *Accounting* action that requires a *VerifySignature* action. When using the *Accounting* action it must be ensured and even be proven that a certain consumer has invoked the component in order to bill him for the usage. This can be achieved with signature verification.
Composition Implications: If there is a requires interdependency for a selected NFA, then the selection of NFAs will be invalid as long as the modeler has not also chosen to select the missing required NFAs.

Precedes: *Precedes* is an ordering interdependency which assumes the action that is the source of a *precedes* interdependency (preceding) must be applied before the target action (preceded). The *precedes* interdependency does not imply a requirement between preceding actions; i.e., if the preceding or preceded actions are not used at all there is no constraint violation. An example for *precedes* is an *Authenticate* action that must precede an *Authorize* action, because the identity of the user must be known before the *Authorize* action is able to decide whether the user has access rights. However, using only *precedes* would not be sufficient in this case, because *Authorize* could be executed without *Authenticate*. Hence, a *requires* interdependency should be used additionally.

Composition Implications: Causes an order restriction in the NFA composition. If the order is wrong because the preceding action is executed after the preceded action, or if both are executed in parallel, an error will be raised.

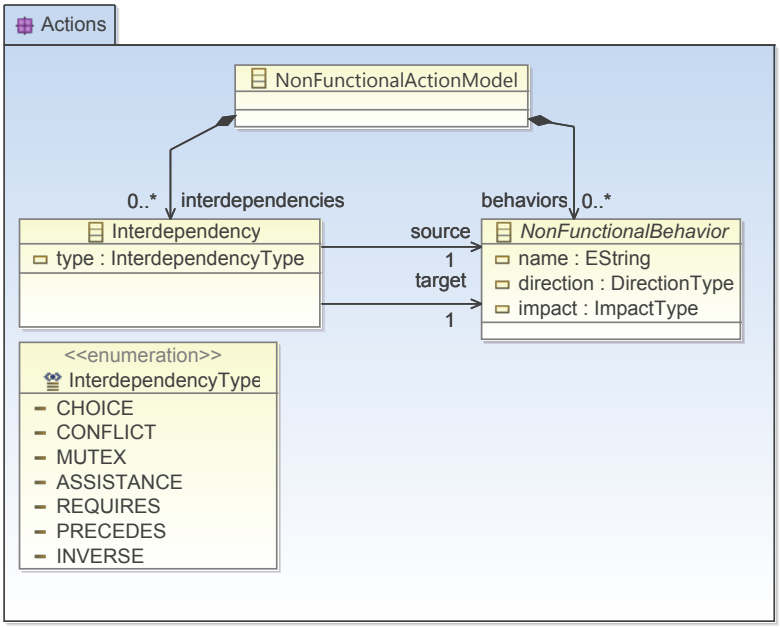

Figure 3.5: Metamodel of Interdependencies

Inverse: The effect of one action may be the inverse of another action. Applied to the same subject, the effect of both actions would compensate. Hence, two inverse actions should rather be applied to two different parties such as application consumer and application provider. For example the *Encrypt* action is inverse to the *Decrypt* action because it encrypts outgoing messages whereas *Decrypt* decrypts incoming messages. Hence, an application consumer should apply *Encrypt* before sending the message and the application provider should apply *Decrypt* after receiving this message.

Modeling Implications: The *inverse* interdependency can be used to infer the inverse actions that must be applied when a consumer wants to stay interoperable with a component that makes use of NFAs. For example, if the application decrypts incoming messages, the consumer must

encrypt all messages sent to this application. Figure 3.5 shows the metamodel for the interdependencies. An *Interdependency* is always defined between two *NonFunctionalBehavior* elements such as *NonFunctionalAction*.

Action Notation: Non-functional actions are notated as arrow symbols showing the most important properties (depicted in Figure 3.6). The name property is separated from the *Impact*, *ImpactPart*, *Direction* and *RealizedAttribute* property by a separator line. Two actions can be connected with an unidirectional line representing the interdependency. The type of the interdependency is annotated as a text near the connection line.

Figure 3.6: Action Notation

3.2.3 Action Composition

When more than one action applies to the same functional subject, the order or control flow of actions must be specified. Otherwise, unanticipated effects may occur because different orders of actions may result in different effects. As already shown in the previous chapter, the desired order often depends on the concrete scenario and requirements. Thus it must be specified explicitly. To encapsulate the action order in a reusable entity which is applicable in different scenarios, NFComp introduces the concept of a non-functional activity[1] which is a container for a concrete order or control flow definition. The language for the order definition is a subset of BPMN2 [67]. A non-functional activity specifies exactly one *Start Event* which defines the entry or starting point of the activity. One or more *Stop Events* can be used to trigger the termination of the whole activity. Furthermore, the non-functional activity defines a special BPMN2 *Activity* called *Non-Functional Task*. A *Non-Functional Task* executes a particular non-functional action in a process context and is used instead of the common BPMN2 activities such as tasks and subprocesses.

To model a sequential execution order, for example, two *Non-Functional Task* elements can be interconnected by a BPMN2 sequence flow connection, which means that the actions are executed one after another by the non-functional tasks.

NFAs specified during the *Action Definition Phase* can be invoked from BPMN *FlowNode* elements which can be connected with other *FlowNode* elements through *SequenceFlow* connections. However, the use of the BPMN language facilitates not only the specification of static, but also that of dynamic, control flow, which is in fact required in particular scenarios, e.g., when

[1] The name activity is derived from activities and actions in UML2

some of the specified NFAs should be executed depending on a particular runtime condition. An example would be a non-functional activity with two *Encrypt* actions, each with a different encryption algorithm (weak and fast vs. strong and slow for example). In a banking scenario, a message with the amount to be debited could be sent to a service which requires confidentiality realized by encryption. Depending on the *amount* parameter of this message, either the strong or the weak *Encrypt* action should be used. In order to model this scenario, BPMN2 gateways in combination with guard conditions on the sequence flow connections can be used. There are three types of gateways supported; *XOR* for exclusive execution (either a or b is executed), *OR* (a and b, a or b) and *AND* for parallel execution (a and b are executed in parallel). The gateways and sequence flow is shown in Figure 3.8. The meaning of the gateway symbol is thereby as follows: *XOR* = X, *AND* = + and *OR* = O.

Not only control but also data flow can be modeled in a non-functional activity. The BPMN *DataItem* can be used to represent any kind of data. This data can be consumed or produced by a *Non-Functional Task* and hence by an action. The consumption, respectively production, of the data is represented by incoming, respectively outgoing, data associations.

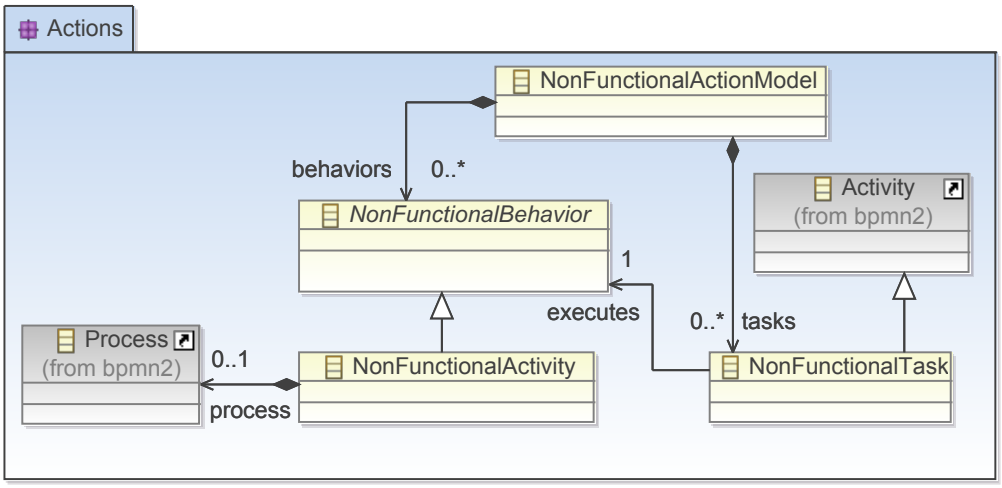

Figure 3.7: Metamodel of Non-functional Activities

Input and output artifacts: The input for the *Action Composition Phase* is the action model conforming to the action metamodel shown in Figure 3.4 and 3.5. The output is the composition model which is an instance of the extended action metamodel shown in Figure 3.7. It additionally introduces the concept of *NonFunctionalActivity*, which is another specialization of *NonFunctionalBehavior*. The *NonFunctionalActivity* contains a BPMN *Process* defining the control and data flow. Furthermore, the *NonFunctionalTask* concept is introduced which executes exactly one *NonFunctionalBehavior* such as *NonFunctionalAction* (depicted in Figure 3.4) or *NonFunctionalActivity* (Figure 3.7). This allows the definition of nested activities which are activities within activities leading to a similar concept as subprocesses in BPMN2. Nested activities foster the reuse of once defined control flow logic.

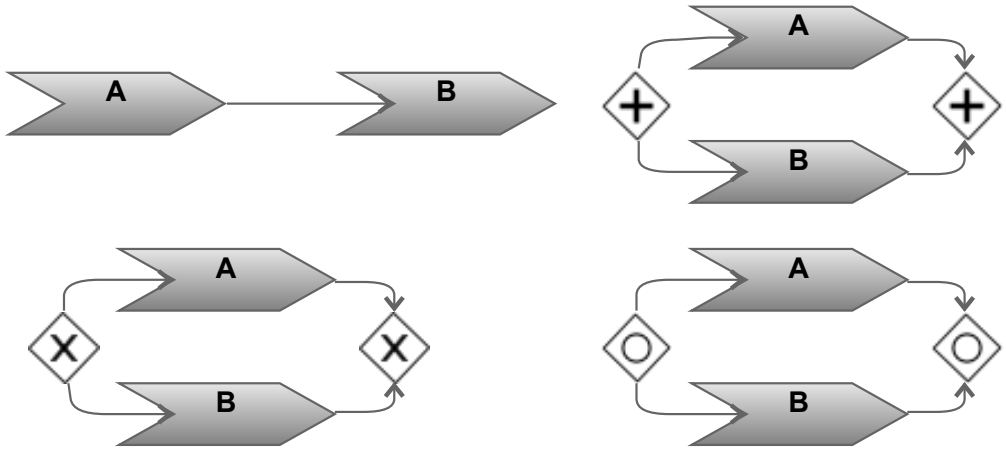

Figure 3.8: Notation for Control Flow with Non-Functional Tasks

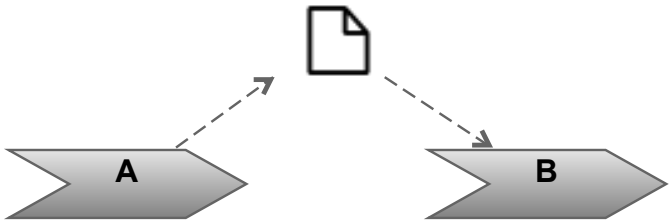

Figure 3.9: Notation for Data Flow Between Non-Functional Tasks

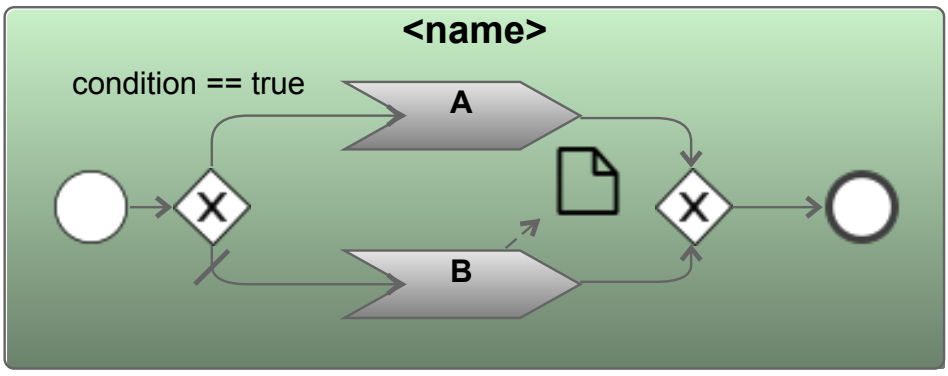

Figure 3.10: Notation for a Non-Functional-Activity

Composition Notation: Figure 3.8 gives an overview on the supported BPMN2 control flow elements: non-functional tasks, sequence flow, XOR, AND and the OR gateway. Figure 3.9 depicts the dataflow using a data item. In Figure 3.10 the notation of a non-functional activity is shown, which is a rounded rectangle showing the name property with bold font at the top and the control and data flow elements below. A guard condition is also depicted as a text annotation above a sequence flow, and a default sequence flow is used to represent the default case when all conditions of all flows connected with the same gateway evaluate to false. If a non-functional task executes an action it will be represented by the already known action symbol. If it executes an activity, it will be represented by the non-functional activity symbol instead.

3.2.4 Action to Application Mapping

The *Action to Application Mapping Phase* defines a model to map the actions and activities to the functional part of the application, i.e., the components implementing the business logic. Applying the non-functional behavior to the functional application enhances the business logic with non-functional concerns to support the desired non-functional requirements. The challenge in this phase is to allow fine-grained as well as coarse-grained mappings depending on the available information. For example, in most component-based approaches, interfaces provide several operations which accept different input and output parameters. The mapping can be performed mainly by the application provider. However, she may need support from non-functional domain experts as well.

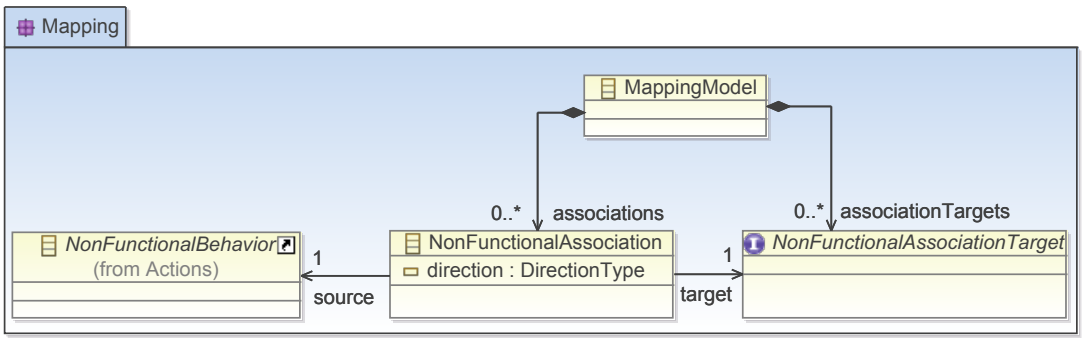

Figure 3.11: Metamodel for the Application Mapping Model

Input and output artifacts: The input for the *Action to Application Mapping Phase* is the action model, the composition model (optional, if only individual actions need to be mapped) and the WSDL files(s) of the service(s) to be mapped. The output is the mapping model conforming to the metamodel shown in Figure 3.11. The metamodel cannot yet be instantiated since it does not define any concrete *NonFunctionalAssociationTarget* elements. However, the metamodel defines a *MappingModel* container element which contains the not-yet-defined targets and the *NonFunctionalAssociation* elements. The *NonFunctionalAssociation* defines a direc-

tion attribute of type *DirectionType*, which has already been defined in the action metamodel (see Figure 3.4).

No notation is available for this general phase; the notation depends on the concrete component-based technology.

3.2.5 Action Realization Mapping

The *Action Realization Mapping Phase* defines the mapping of non-functional actions to software components implementing the algorithm or behavior of these actions. This mapping must be done by a domain expert who has deep knowledge in a particular non-functional domain such as Security or Performance, for example. This phase is necessary as prerequisite for the next phase *Code Generation*.

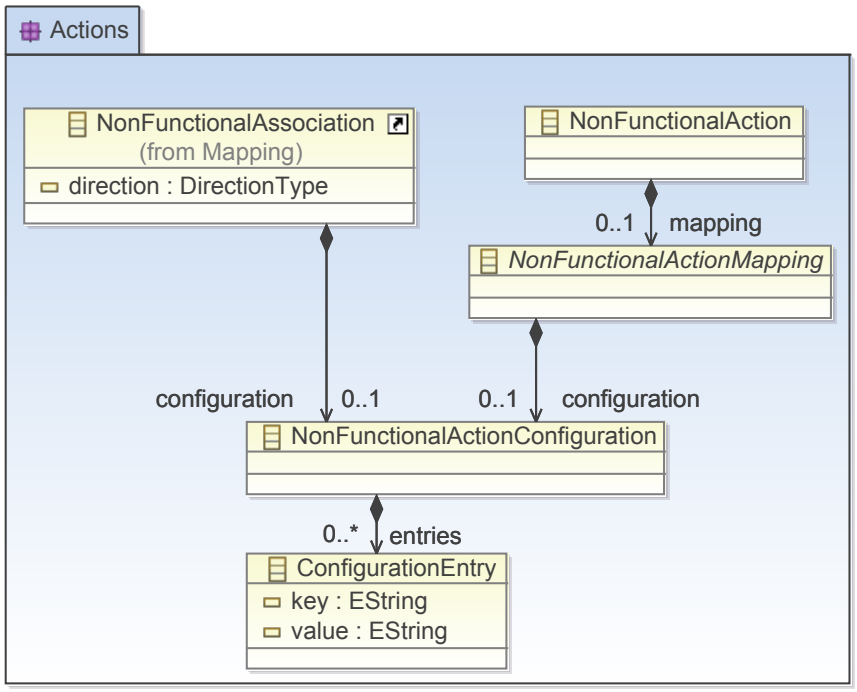

Figure 3.12: Metamodel for the Action Realization Mapping

Input and output artifacts: The input for the action realization mapping is the action model conforming to the action metamodel shown in Figure 3.4 and 3.5. This phase extends the action metamodel by the abstract concept of the *NonFunctionalActionMapping* which is associated with a *NonFunctionalAction* (see Figure 3.12). The *NonFunctionalActionMapping* is abstract, describes the mapping to a component realizing the action and must be extended when a concrete component-based technology is used. Generally, it can be assumed that a certain *NonFunctionalActionConfiguration* can be assigned to a *NonFunctionalActionMapping*. Each configuration is constituted by a set of *ConfigurationEntry* elements which are key-value-pairs of type *EString*.

A more precise type cannot be assumed due to the heterogeneity of the non-functional actions. In order to improve reusability of actions, a particular configuration can be overridden by assigning a configuration to a *NonFunctionalAssociation*. This allows the mapping of configurations to specific appliances of actions to components, e.g., the same *Encrypt* action can be used with different encryption algorithms for different components. If there were no such mechanism in place, it would be necessary to model a separate action for each encryption algorithm. This would be too fine-grained, because the properties of those actions would be mostly identical.

No particular notation for the mapping is available because it is a technical detail which is heavily table structured. Thus, a textual editor such as a property sheet can be used to maintain the mapping with the configuration parameters.

3.2.6 Code Generation

When a code generator is implemented, it must be clear which software component will realize which action at runtime. Otherwise it cannot transform the model into executable code. Hence, this phase cannot be described precisely, as yet. However, in the application of NFComp to web service technology, this phase will be described in more detail.

3.3 Towards Conflict-Free Action Compositions

The composition of actions in the *Action Composition Phase* is a complex task because knowledge from different non-functional domains is required in order to understand the impact of different NFAs. Thus, it is helpful to define a formal interdependency model that supports the identification of invalid composition definitions at design time. The goals of this section are, more specifically,

- to enrich the action model by discovery of cross-domain interdependencies through analysis of the data impact of NFAs,
- to use the interdependency model to provide support for composing NFAs by
  - visualizing constraint violations in the composition,
  - suggesting conflict resolution strategies for violated constraints,
  - introducing a guided modeling procedure.

3.3.1 Formalizing the Interdependency Model

3.3.1.1 Tasks and Actions

Let $\mathcal{A}$ be a set of NFAs, and let $\mathcal{T}$ be tasks, each executing an NFA. Then, $executes \subseteq \mathcal{T} \times \mathcal{A}$ is the relation defining which task executes which action. In contrast to actions, tasks are part of a specific execution context (i.e., a process) and therefore have a well-defined execution order.

This relation can be compared to the relation between BPMN [67] service tasks and the services called by these tasks. More details of the composition model can be found in Section 3.3.2.1.

3.3.1.2 Interdependencies

As introduced in Section 3.2.2.2 there are 7 interdependency types namely *choice*, *conflict*, *requires*, *precedes*, *assists*, *mutex* and *inverse*. However, with respect to control flow in a non-functional activity these types can be classified into three equivalence classes:

$Class_{exclusion} = \{mutex, inverse, choice, conflict\}$
$Class_{dependency} = \{requires, assists\}$
$Class_{precedes} = \{precedes\}$

All interdependency types in $Class_{exclusion}$ define a certain level of exclusion between the actions they are defined for. $Class_{dependency}$ defines some level of dependency between the action, for example, a strong *requires* or a weaker *assists*. *Precedes* defines restrictions on the ordering of actions and constitutes its own equivalence class.

Hence, it is possible to focus on the strongest representatives of the given interdependency types. Let $\mathcal{I} := mutex \cup requires \cup precedes$ be the set of interdependencies between actions and tasks. More specifically it is

- $requires = \{(x_1, x_2) \in (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{T} \times \mathcal{T})$ | Execution of x_1 requires the execution of $x_2\}$
- $precedes = \{(x_1, x_2) \in (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{T} \times \mathcal{T})$ | If x_1 and x_2 are both executed, x_1 must be executed before $x_2\}$
- $mutex = \{(x_1, x_2) \in (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{T} \times \mathcal{T})$ | Execution of x_1 excludes the execution of $x_2\}$

From a given set of interdependencies, further interdependencies can be inferred by symmetry and transitivity. *Mutex* is symmetric, i.e., $mutex(x_1, x_2) \rightarrow mutex(x_2, x_1)$. *Requires* and *precedes* are both transitive, i.e., $precedes(x_1, x_2) \wedge precedes(x_2, x_3) \rightarrow precedes(x_1, x_3)$ (where $x_1, x_2, x_3 \in \mathcal{A} \cup \mathcal{T}$). In addition to these interdependencies, there are also action properties — as already introduced in Section 3.2.2.1 — that play an important role for the composition. These properties can be categorized into data-related properties and control-flow-related properties. However, only the data-related ones are taken into account for validation since control-flow related properties should be explicitly modeled in the control flow defined by the non-functional activity. A blocking action should be modeled as an exclusive gateway with one branch executing the action and the other terminating the activity. This allows the explicit definition of control flow in BPMN which is more powerful, flexible and simplifies validation and code generation.

3.3.1.3 Data Dependencies

Let $\mathcal{A}$ be a set of actions and $\mathcal{D}$ be a set of data items (which can be of complex type) and $a \in \mathcal{A}$ and $d \in \mathcal{D}$. Then, $\mathcal{P} := \{read, add, remove, modify\}$ is the set of binary relations between actions and data which are called impact types (because they define the impact on data) with the following semantics:

- $read = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a reads data item d$\}$
- $add = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a adds data to data item d$\}$
- $remove = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a removes data item d$\}$
- $modify = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a modifies (and reads) data item d$\}$

Let a and b be actions accessing data item d and let a be executed directly before b, i.e., there is no other action $c \neq a, b$ accessing d executed between a and b. Then, there are 16 possible combinations of impact types to be analyzed. There are 10 combinations that cause or may cause conflicts with respect to their impact on data as shown in Table 3.1. In this table, conflicting combinations of impact types are represented as follows: $-$ conflict, $(-)$ potential conflict, $+$ no conflict, (+) warning, R = reverse order also in conflict. Subscripted numbers indicate the number of the enumeration item which explains the respective conflict.

	read(b,d)	**add(b,d)**	**remove(b,d)**	**modify(b,d)**
read(a,d)	$+$	$-_{2}$	$+$	$+$
add(a,d)	$+$	$-_{6,R}$	$(+_{5})$	$+$
remove(a,d)	$-_{1}$	$+$	$-_{7,R}$	$-_{9,R}$
modify(a,d)	$(-_{4})$	$-_{3}$	$(+_{8,R})$	$(-_{10,R})$

Table 3.1: Conflict Matrix for Impact Types

1. $remove(a, d) \wedge read(b, d)$ Data *d* is removed by *a* before *b* is able to read it.
2. $read(a, d) \wedge add(b, d)$ Data *d* is read by *a* before *b* adds it. Either *d* exists before execution of *a* which would lead to duplicated data by the execution of *b*, or *d* must be added by *b* because it does not exist, so *a* would read non-existing data.
3. $modify(a, d) \wedge add(b, d)$ Data *d* is modified by *a* before *b* adds it. This combination is similar to 2, because $\forall a \in \mathcal{A}.modify(a, d) \rightarrow read(a, d)$.
4. $modify(a, d) \wedge read(b, d)$ Data *d* is modified by *a* before *b* can read it. This could be a potential conflict because *a* modifies the data which causes a state change from d_1 to d_2. It depends which state *b* expects. If *b* expects *d* to be in state d_1, this combination is a conflict.

5. $add(a, d) \wedge remove(b, d)$ Data *d* is added by *a* and then removed by *b*. Removing data directly after adding it makes no sense, but does not cause problems at runtime.

6. $add(a, d) \wedge add(b, d)$ Data *d* is added by action *a* and *b* and hence duplicated.

7. $remove(a, d) \wedge remove(b, d)$ Data *d* is removed by action *a* and *b*. After execution of *a*, data *d* does not exist anymore; hence *b* cannot remove it.

8. $modify(a, d) \wedge remove(b, d)$ Data *d* is modified by *a* and then removed by *b*. Removing data directly after modifying it makes no sense, but does not cause problems at runtime.

9. $remove(a, d) \wedge modify(b, d)$ Data *d* is removed by action *a* and then modified by *b*. This is similar to 1 because $\forall a \in \mathcal{A}.modify(a, d) \rightarrow read(a, d)$.

10. $modify(a, d) \wedge modify(b, d)$ Data *d* is modified by *a* and then modified again by *b*. As modify also reads data, this is similar to 4.

A discussion of the strategies for resolving these conflicts is given in the following. Since the strategies can only be applied when the execution order of actions is already known, it is assumed that tasks x and y are executing conflicting actions a and b which access the same data item d such that x is executed before y and there is no task z between them which also accesses d. The first five conflict situations are resolvable by inverting the execution order of tasks *x* and *y* because the reverse order causes no data conflicts as can be seen in Table 3.1. For these combinations, the *precedes* interdependency is added to the set of interdependencies; i.e., $precedes(y, x)$ in this case. No other combination can be resolved by reordering because the inverse order may also cause conflicts. These conflicts can be resolved either by removing one of the tasks or by executing them exclusively; i.e., by executing either *x* or *y* depending on a particular condition. For all these combinations, *mutex* is added to the set of the existing interdependencies, i.e., $mutex(x, y)$ in this case.

3.3.2 Supporting Action Composition by Validation, Solution Strategies and Conflict-Free Composition Procedure

3.3.2.1 The Composition Model

For the composition of actions, a subset of BPMN2 [67] is used. A model in BPMN2 is a set of nodes and transitions between them. For the validation a simplified process model suffices which is defined as follows. A non-functional activity is a directed graph containing all process elements. It is $activity := (\mathcal{N}, \mathcal{E})$ with the following semantics:

$\mathcal{N} := \{x|node(x)\} \equiv \{x|x \text{ is process node}\}$
$\mathcal{E} := \{(x,y)|transition(x,y)\} \equiv \{(x,y) \in \mathcal{N} \times \mathcal{N}|\text{there is a transition from x to y}\}$
$start := \{x|x \text{ is the start node of the process}\}$, $end := \{x|x \text{ is an end node of the process}\}$
$\mathcal{T} := \{x|task(x)\} \equiv \{x|x \text{ is task node}\} \subset \mathcal{N}$
$\mathcal{G} := \{x|gateway(x)\} \equiv \{x|x \text{ is gateway node}\} \subset \mathcal{N}$
$\mathcal{XOR} := \{x|gw\_xor(x)\} \equiv \{x|x \text{ is xor gateway}\} \subseteq \mathcal{G}$
$\mathcal{OR} := \{x|gw\_or(x)\} \equiv \{x|x \text{ is or gateway}\} \subseteq \mathcal{G}$
$\mathcal{AND} := \{x|gw\_and(x)\} \equiv \{x|x \text{ is and gateway}\} \subseteq \mathcal{G}$
$\mathcal{M} := (\mathcal{T}, \mathcal{XOR}, \mathcal{OR}, \mathcal{AND}, start, end)$

$\mathcal{M}$ is a tuple of sets $M_0, M_1...$ and each node n is exactly in one of its set elements: $(\forall i, j < |M|)M_i \cap M_j = \emptyset$ for $i \neq j$, and $(\forall n \in \mathcal{N})(\exists i)n \in M_i$. Moreover, there is exactly one start node: $|start| = 1$.

3.3.2.2 Identifying Constraint Violations

To identify violations of the given interdependency constraints, a non-functional activity must be checked against each individual interdependency. For each interdependency $i = (a, b) \in \mathcal{I}$, the occurrence and order of actions (or tasks) *a* and *b* in the same execution path can lead to constraint violations depending on the given type. Hence, all possible execution paths through the process graph must be analyzed. The number of these paths depends on the control flow of the process, more specifically on the number of OR and XOR gateways and the number of outgoing sequence flows per gateway. Presuming all gateways are used in sequence and $out : \mathcal{G} \rightarrow \mathbb{N}$ is a function that calculates the number of outgoing sequence flows for each gateway, the number of possible paths for a given number of XOR and OR gateways, respectively, in the worst case is:

$$paths_{xor} = \prod_{x \in \mathcal{XOR}} (out(x)) \qquad paths_{or} = \prod_{x \in \mathcal{OR}} (2^{out(x)})$$

Obviously, the number of possible paths for *OR* is much higher than that of *XOR*. Let *k* be the number of all outgoing sequence flows from *OR* gateways and *traverse* be a function which traverses all possible paths; then the complexity of this function can be estimated using the $\mathcal{O}$-Notation: $traverse \in \mathcal{O}(2^k)$, resulting in exponential runtime complexity. The declarative Prolog [20] language is a good choice for searching defined spaces for possible solutions because it provides very efficient ways to do a depth-first search with backtracking. When traversing the search tree, backtracking allows one to remember potential candidates for solutions at each tree node. If Prolog finds out that a particular candidate cannot satisfy the problem, it will drop it and try to solve the problem using one of the remaining candidates.

3.3.2.3 Using Prolog to Find Violations and Counter Examples

In order to use Prolog for constraint checking, the BPMN models defining the control flow of a non-functional activity need to be imported and translated into Prolog. This can be achieved by transforming activities into a fact data base which contains a set of predicates. Hence, the following predicates have been defined: *start(x)*, *end(x)*, *task(x)*, *gw_xor(x)*, *gw_or(x)*, *gw_and(x)*, *node(x)*, *transition(x,y)*, and *executes(x, a)*. The last predicate indicates that the task node x of the process executes action a. Tasks and actions are distinct concepts to cope with processes in which different task nodes execute the same action. The constraints given by the interdependency model form the rules that should apply to the fact base. However, the challenge for defining these rules is that the violation of a particular rule should not result in a simple boolean true or false decision but should also provide counter examples to give the modeler of non-functional activities constructive feedback.

With its backtracking concepts, Prolog allows for obtaining all values for which a certain predicate evaluates to true. Therefore, a predicate with parameters A, B, X, Y, and P can be defined for each interdependency type so that the predicates are true if and only if P is a counter example for the respective interdependency regarding the actions A and B which are executed in nodes X and Y, respectively. Within a query, Prolog distinguishes between constants and variables: If a variable is used for a certain parameter of a predicate, Prolog will search for values of this variable fulfilling the predicate whereas constant parameters restrict the search space. It can be assumed to have a constant list of interdependencies between actions and tasks. In case of an action interdependency, the appropriate predicate can just be applied for the respective interdependency type to constants for A and B and variables X, Y, and P for Prolog to yield possible counter examples as solutions for X, Y, and P. In case of a task interdependency, the procedure is analogous, but then the task constants for X and Y are used.

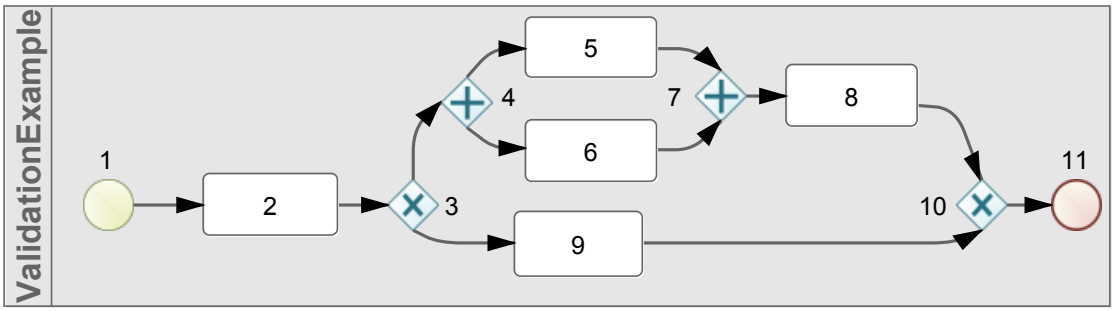

Figure 3.13: Example BPMN for Validation

Regarding the structural representation of counter examples, the following concepts of paths in BPMN processes have been introduced: A **Plain Graph Path** (PGP) from X to Y is a simple path in the BPMN process graph as known from graph theory of directed graphs. It is represented as the list of nodes contained in the path (in Figure 3.13, for example, 1,2,3,9,10,11 is a PGP). A **Block Path** (BP) is a PGP in which nodes between opposite gateways are left out.

BPs can only exist between two nodes if they have the same parent node. The term *parent node* refers to a tree representation of the BPMN process nodes in which the parent node of each node is the gateway in which it is contained, or, if it is not contained in any gateway, an imaginary root node. For each pair of nodes *X*, *Y* with the same parent node, there is exactly one BP between these nodes if a PGP from *X* to *Y* exists (in Figure 3.13, for example, 1,2,3,10,11 or 4,7,8 is a PGP where node 3 represents the whole block between 3 and 10 and 4 represents 4,5 and 6). A **Block Execution Path** (BEP) is a BP where each gateway node is replaced with a pair (X, P). *X* is the replaced gateway node itself, and P is a list of BEPs that are executed in parallel starting from the gateway *X*. BEPs respect gateway semantics; e.g., the number of paths starting from an XOR gateway is always 1. A BEP is therefore an appropriate representation of a concrete execution of the BPMN process; for example, Gateway 4 in Figure 3.13 will be represented by [4, [[5], [6]]], and Gateway 3 as [3, [[[4, [[5], [6]]]]]]. Particularly, BEPs are used to represent counter examples in the aforementioned rules. A BEP is called complete if it begins with a start node and ends with an end node. Specifically, the following rules for the interdependency types have been defined, each starting with *ce* as an abbreviation for *counter example*. Assuming that *P* is a complete BEP, then it is

- $ce\_conflicts(A, B, X, Y, P)$ is true if *P* contains both *X* and *Y* which in turn execute the actions *A* and *B*, respectively.
- $ce\_precedes(A, B, X, Y, P)$ is true if both action *A* and action *B* are executed in *P*, but task *Y* which executes *B* is in this path not guaranteed to be preceded by another task which executes *A*. *X* is just any task executing *A* in this path. Intuitively, this predicate is true if action *A* is not guaranteed to be executed before *B*. This is the case if *B* appears sequentially before the first *A*, or if *A* and *B* are executed in parallel paths of the same gateway.
- $ce\_requires(A, B, X, Y, P)$ is true if *A* is executed by *X* in *P*, but there is no task executing *B*. By convention, *Y* is set to 0.

Each of the predicates can be used to obtain counter examples by defining constants for *A* and *B* and using variables for *X*, *Y*, and *P* for which Prolog will try to find instances which make the predicate true. For this purpose, Prolog iterates over all complete BEPs and tasks *X*, *Y* executing *A*, *B* and returns the first combination fulfilling the respective predicate. If no counter example exists, no solution will be found.

3.3.2.4 Conflict Resolution

After identifying interdependency violations in a non-functional activity, strategies for solving these conflicts should be defined. Table 3.2 shows the different strategy classes. The difference between *rearrange* and *move* is that with *rearrange* an action will not move from one execution branch to another. As can be seen in the table, all resolution strategies may impose new conflicts.

Resolution Strategy	Solves	May Introduce
Remove Action	Mutex, Prec	Req
Insert Action	Req, Prec	Prec, Mutex
Rearrange Action	Prec	Prec
Move Action	All	All
Transform Gateway	All	All

Table 3.2: Resolution Strategies: Interdependency Conflicts Solved and Introduced

To avoid these undesired side effects it has been analyzed under which conditions a certain strategy can be applied safely. Before *remove action* can be applied safely to action a, for instance, it must be checked whether there is a *requires* interdependency from any other action to a. In general, both safe and potentially unsafe strategies are distinguished.

3.3.2.5 Conflict-Free Composition Procedure

As discussed in the previous subsection, it is complex to provide a validation mechanism with automatic conflict resolution. Usually, human intervention is required at a certain point. It is even harder to propose a complete conflict-free activity because of the possibly small sets of given interdependencies. In order to combine the power of the presented validation approach with the ability of human non-functional domain experts to compose activities, a new guided modeling procedure is presented in the following. The idea is that a composition tool can be used to model a start event and the tool then proposes the next valid steps, always leading to correct processes with respect to interdependency constraints. For this purpose, the Prolog implementation must be extended in the following way: It should take a predefined set of candidate actions and the BPMN node from where to insert the next action as input. The output should be a list of valid actions which, when inserted at this point, would cause no interdependency violations.

The concrete process for obtaining a list of valid actions A to be proposed for insertion at a certain position consists of the following steps: (1) Virtually extend the current (incomplete) BPMN process by adding a placeholder task x at the position where the user wants to insert a new element. Also, for each node of the process without an outgoing edge, add an edge to the end event which is newly created if necessary. This allows a complete BPMN process enclosed by a start and an end event which the Prolog program is able to process. (2) Send a query to Prolog to obtain all actions I which would violate a constraint if they were executed by x. This query is based on the Prolog model of the BPMN process such as used during validation and, additionally, on the Prolog model of all interdependencies relevant for the BPMN process. These are expressed in terms of a list of Prolog facts based on Prolog predicates $precedes$, $requires$, and $conflicts$, each having two parameters defining the actions or tasks between which the respective interdependency exists. The query also contains the list of candidate actions C to be tested at the position of x. (3) NFComp's Prolog program then consecutively assumes x to

execute each of the candidate actions and returns the list I of those for which at least one of the defined interdependencies is violated. (4) The actions $A := C \setminus I$ are proposed to the user for insertion at the specified position. By definition of the proposed actions A, the obtained process after insertion of one of them would never result in a new constraint violation. This composition technique considerably facilitates the definition of conflict-free compositions.

3.4 NFC Composition in a Black Box View of Web Services

In this section NFComp is applied to a concrete component-based technology, namely web services. Web services can be regarded from different views: black, gray and white box. This section will take the black box view and revisit the different phases introducing additional concepts for the application of NFComp to web services. More specifically, **Action to Application Mapping**, **Action to Component Mapping** and **Code Generation** phases are concretized for this view.

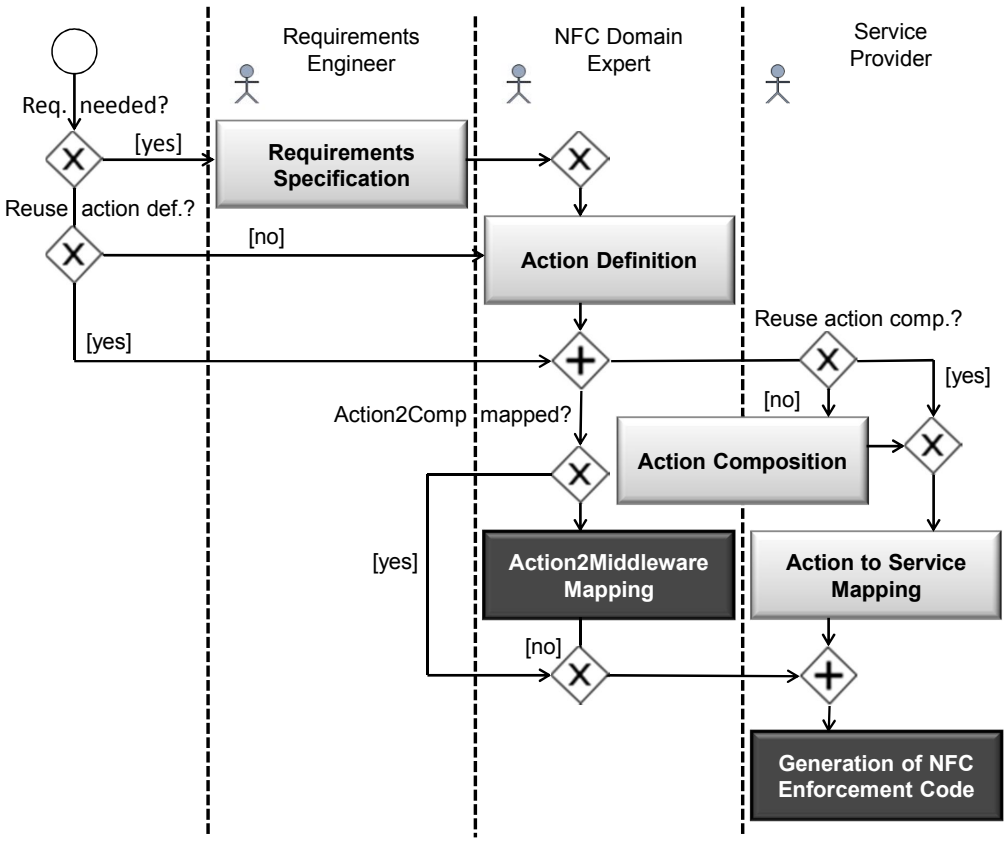

Figure 3.14: NFComp Process for Web Services

Figure 3.14 shows the NFComp process model for web services. In this model, NFComp is depicted as a process similar to BPMN emphasizing optional phases and parallelism. In the following one running example per view will be presented to see the explained concepts instantiated. In these examples, the Eclipse-based tool set provided by NFComp is used. The tool

set comprises a set of Graphiti[2]-based editors. There is one dedicated editor for *Requirements Specification*, one for *Action Definition* and *Action to Middleware Service Mapping*, one for *Action Composition* and one for *Action to Service Mapping*.

3.4.1 Requirements Specification

The **Requirements Specification Phase** for Web services from the black box view is basically the same as described in Section 3.2.1. There is no need to introduce additional extensions because requirements are described in a platform-independent manner.

Running Example

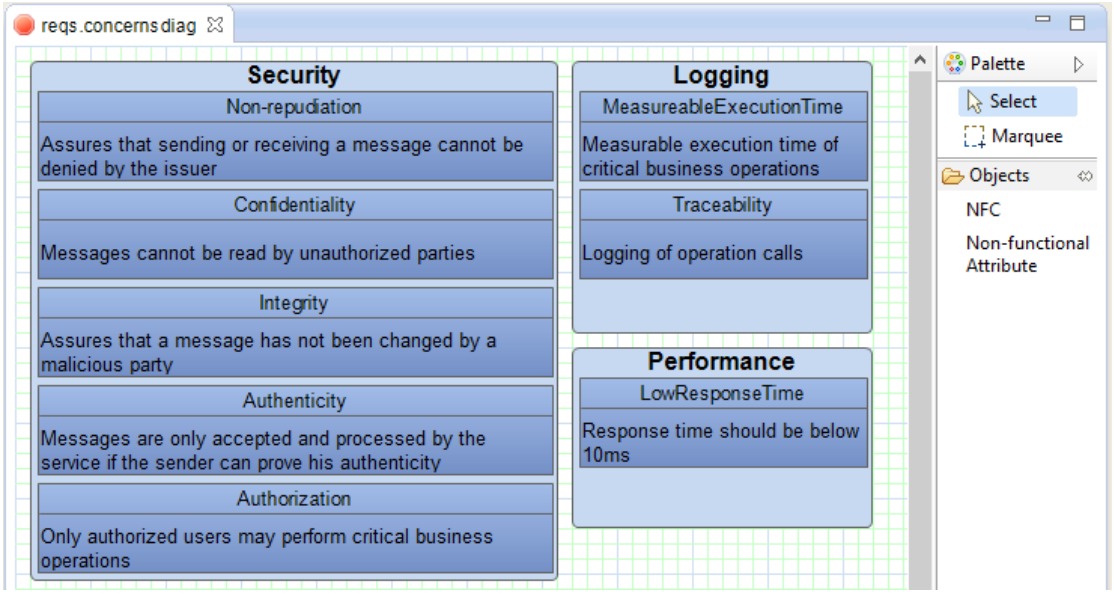

Figure 3.15: The Requirements Editor: Modeling Concerns and Attributes

For the running example on web service from the black box view, an enterprise is assumed which has transformed its IT assets into a set of commercial web services. Furthermore, it is assumed that these web services have been implemented in different programming languages, for instance due to constraints introduced by legacy systems. During the development of the web services, non-functional concerns have been ignored on purpose as they should be strictly separated from the business functionality of the services. Hence, depending on the respective features the service provides, different non-functional requirements have been identified. For example, since the services are commercial, authentication and authorization are required to restrict the access to the services only to registered customers. To bill the customers based on the service usage, an accounting mechanism is required as well as support for non-repudiation and integrity of the messages. Furthermore, the company decided to log messages in the early

[2] http://www.eclipse.org/graphiti

introduction phase and to monitor the response time of their services. Another requirement is that the response time should be as low as possible. The requirements engineers who determined the requirements for the web service, model them using the Requirements Editor shown in Figure 3.15.

3.4.2 Action Definition

Applying the **Action Definition Phase** to a concrete, component-based technology such as web services allows the assumption of a particular message format — SOAP — and interface — WSDL. A non-functional action has a property *impact* already known from the general concepts introduced in Section 3.2.2. This property describes what the action does with the message, for example adding information, removing it or modifying particular message parts. In Section 3.3.1, conflicting data dependencies have already been identified. As already stated, conflicts only occur when the same data item is affected by two distinct actions. This kind of information is not yet part of the action metamodel because the message format has been abstract so far. When assuming SOAP as a concrete message format, more precise statements can be made about the affected parts. The additional action property *Impact Part* allows definition of the target that is affected by the action. Because SOAP is based on XML, XPath is a suitable expression language to describe in detail which parts of the message are affected.

Running Example

The security, billing/accounting, performance, monitoring/logging, and general web service experts of the company transform the requirements into non-functional actions capable of fulfilling the requirements. The security expert knows how to support the security requirements and defines the following actions using the action editor: *Authenticate* (for authenticity), *Authorize*, and *VerifySignature* (for non-repudiation and integrity). Having defined these actions, the security expert identifies possible interdependencies between them. She defines that *Authenticate* must precede *Authorize* and that *Authorize* requires *Authenticate*. The modeled actions are shown in 3.16. Furthermore, the security expert uses XPath expressions to describe the data items of the SOAP message which the actions have an impact on; e.g., *Authenticate* will read the *UsernameToken* which is part of the *Security* XML tag of the SOAP message header. A summary of all actions which have been defined by all experts can be found in Table 3.3. Table 3.4 summarizes all interdependencies that have been discovered by the experts. In the example, one can see that most of the interdependencies are only discovered between actions defined by the same expert. Cross-concern interdependencies are only defined by the accounting expert who knows that due to legal issues she must prove that a service invocation has really been caused by a certain customer (see Figure 3.17). Cross-concern interdependencies are generally hard to find because the experts must understand and analyze all actions of all non-functional domains.

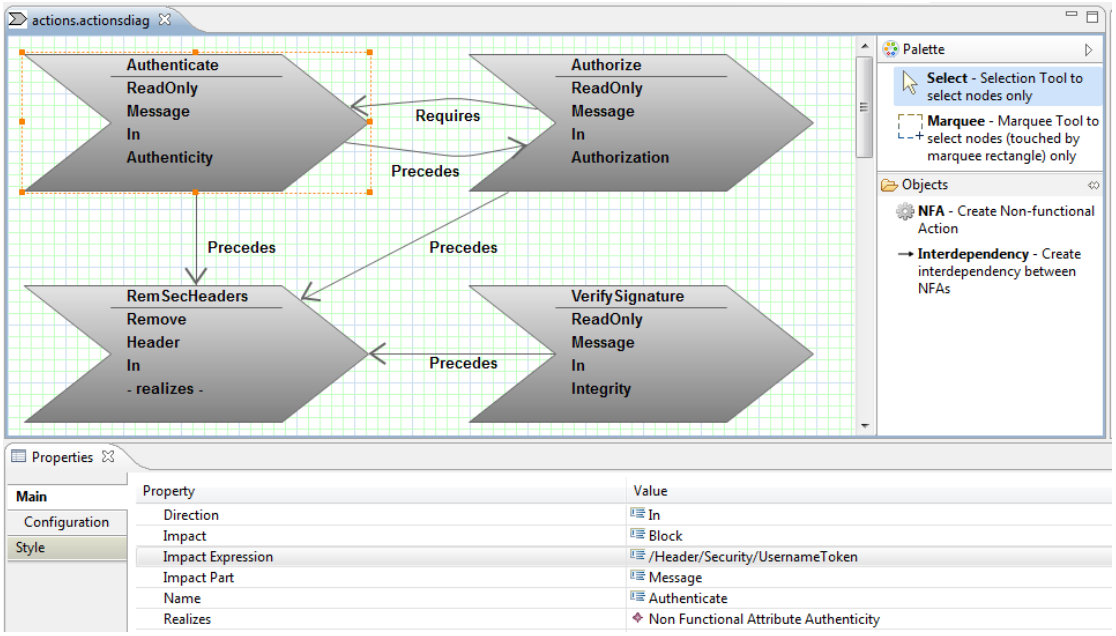

Figure 3.16: The Action Editor: A Security Expert Has Modeled the Actions and Their Properties and Interdependencies

Action	Expert	Impact (XPath)
Authenticate	Security	Read(/Header/Security/UsernameToken)
Authorize	Security	Read(/Header/Security/UsernameToken)
VerifySignature	Security	Read(/Header/Security/BinarySecurityToken, /Security/Signature)
RemSecHeaders	Security	Remove(/Header/Security)
Log	Log/Mon.	Read(/Message//*)
StartTimer	Log/Mon.	None
StopTimer	Log/Mon.	None
ReadFromCache	Perform.	Read(/Body//*)
RemoteAccounting	Acc./Bill.	Read(/Body//*)
LocalAccounting	Acc./Bill.	Read(/Body//*)
RemAllHeaders	General	Remove(/Header//*)

Table 3.3: Actions and Their Impact

Interdependencies
precedes(Authenticate, Authorize)
requires(Authorize, Authenticate)
precedes(StartTimer, StopTimer)
requires(StopTimer, StartTimer)
requires(LocalAccounting, VerifySignature)
requires(RemoteAccounting, VerifySignature)
mutex(RemoteAccounting, LocalAccounting)
precedes(Authenticate, RemSecHeaders)
precedes(Authorize, RemSecHeaders)
precedes(VerifySignature, RemSecHeaders)

Table 3.4: Explicitly Defined Interdependencies

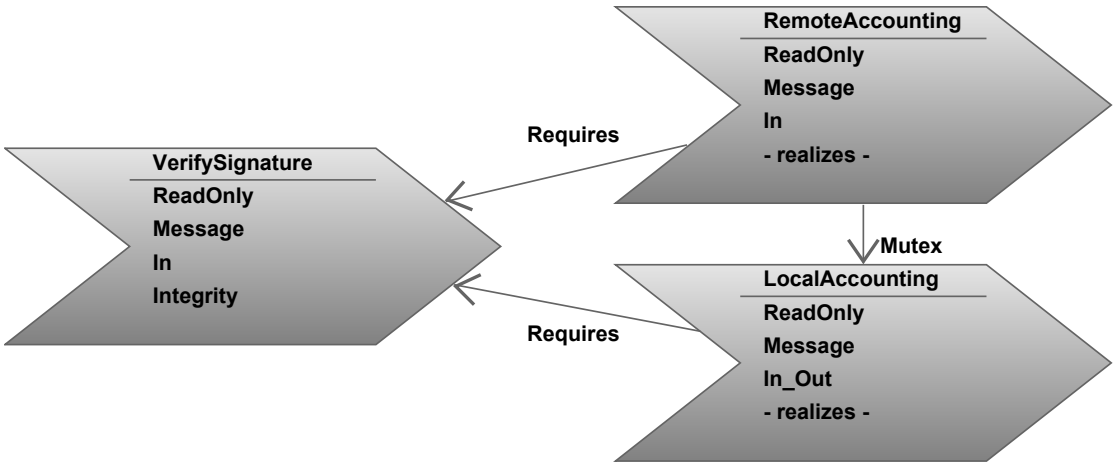

Figure 3.17: A Performance Expert Has Modeled Her Actions and Imported a Security Action for the Specification of Cross-Domain Interdependencies

3.4.3 Action Composition

When more than one action applies to the same subject, their order or control flow must be specified. The reason is that different orders of actions may cause different effects. For example, when an *Encrypt* and a *Log* action are composed, there are two possible variants. In the first variant the *Log* action precedes the *Encrypt* action, which causes plain messages to be written to a log file. In the second variant the *Encrypt* action precedes the *Log* action and thus encrypted messages are logged. Both variants are possible: In the first variant the advantage is that an administrator can look into the log files and analyze/trace the service invocations, whereas in the second variant the message contents are confidential and cannot be read by system administrators. The order that should be used cannot be determined automatically in most cases but depends on the concrete scenario and requirements. For example if there are high confidentiality requirements, the second alternative is the one that should be chosen; if not, then the first alternative suits better because it provides better traceability.

3.4.3.1 Running Example

In this modeling phase, the non-functional activity is created with the composition editor (shown in Figure 3.18, 3.19 and 3.20) by importing the action definition and dragging the available actions from the palette into the activity. An action is executed by a special BPMN task (the non-functional task, arrow symbol), and additional gateways and sequence flow elements can be used to define the control flow.

When a concrete execution order at the task level is given, the previously defined interdependencies can be enriched by additional task interdependencies derived from the data dependencies: Possible data conflicts are identified by an intersection of the XPath expressions defining the data items that are affected by an action (shown in Table 3.3). If there is at least one node that both expressions have in common, the impact types are compared to each other. If, for instance, in the given process a task executing the *RemAllHeaders* action precedes another task executing an action accessing parts of the message header, there is a *remove-read* conflict between the two tasks. This data conflict can be resolved by introducing a *precedes* constraint upon these tasks. Another data conflict can be found when looking at the *RemAllHeaders* and the *RemSecurityHeaders* actions. The latter removes a subset of the data that *RemAllHeaders* removes. This is a *remove-remove* conflict which can be solved by introducing a *mutex* interdependency between the tasks executing these actions. The interdependencies that can be inferred using this mechanism have been collected in Table 3.5.

Implicit Interdependencies	**Implicit Interdependencies**
precedes(Authenticate, RemAllHeaders)	precedes(Log, RemAllHeaders)
precedes(Authorize, RemAllHeaders)	precedes(Log, RemSecHeaders)
precedes(VerifySignature, RemAllHeaders)	mutex(RemSecHeaders,RemAllHeaders)

Table 3.5: Implicit Interdependencies

3.4.3.2 Validation in the Running Example

Validation for a modeled action composition can be started by pushing the *Validate* button in the composition editor. Internally, the process and interdependency data, which is saved as an Ecore model, is transformed into Prolog facts and processed by the Prolog program. A list of all problems is shown in the problems view: a violation of *precedes* between *Authorize* and *Authenticate*, a violation of *mutex* between the accounting actions, and two *requires* violations due to the lack of *VerifySignature*. The selected problem is highlighted (see Figure 3.19). Moreover, so-called quick fixes are available via the context menu of each problem. In the example, the modeler can, for example, remove one of the tasks that are executing mutually exclusive actions or introduce an XOR gateway, for example.

In the approach presented above, the modeler gets feedback only when she triggers the validation. However, it is usually better to avoid these mistakes during the modeling process. This is supported by the guided modeling procedure. Using this procedure, the user starts modeling and a context pad shows all available actions she can add next as shown in Figure 3.18. In the context pad (provided by the Eclipse Graphiti framework), the next valid actions are shown, for example after choosing the *LocalAccounting* action; the *RemoteAccounting* action is not available anymore except in another branch of an XOR gateway.

Figure 3.20 shows the resulting valid non-functional activities which have been created by the service provider. There are two activities: one for incoming messages and one for outgoing ones. The incoming activity is much more complex because several actions need to be composed. Firstly, the *Log* action is executed. Then, the *VerifySignature* action and the *Auth** activity are executed in parallel. *Auth** is a nested activity and defined as a sequence of *Authenticate* and *Authorize*. The parallel execution of *RemSecHeaders* and *RemoteAccounting* follows, and finally the *StartTimer* action is executed. *OutgoingActivity* defines the sequential execution of *StopTimer* and *Log*.

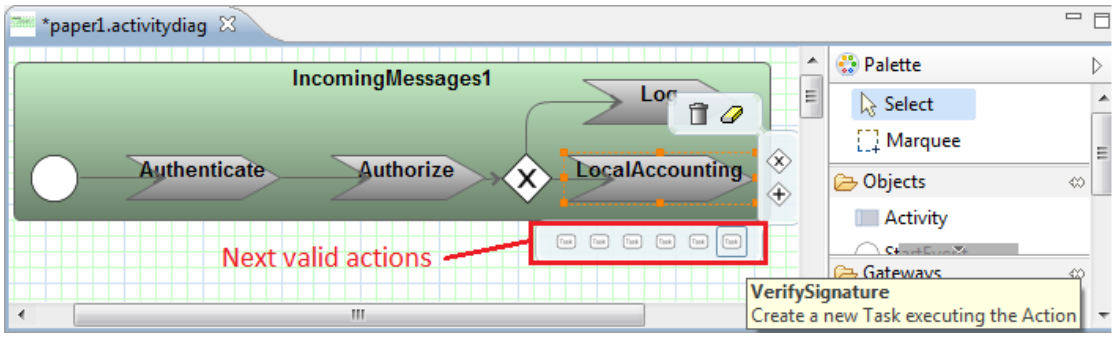

Figure 3.18: The Composition Editor: Guided Composition Proposes Next Valid Actions

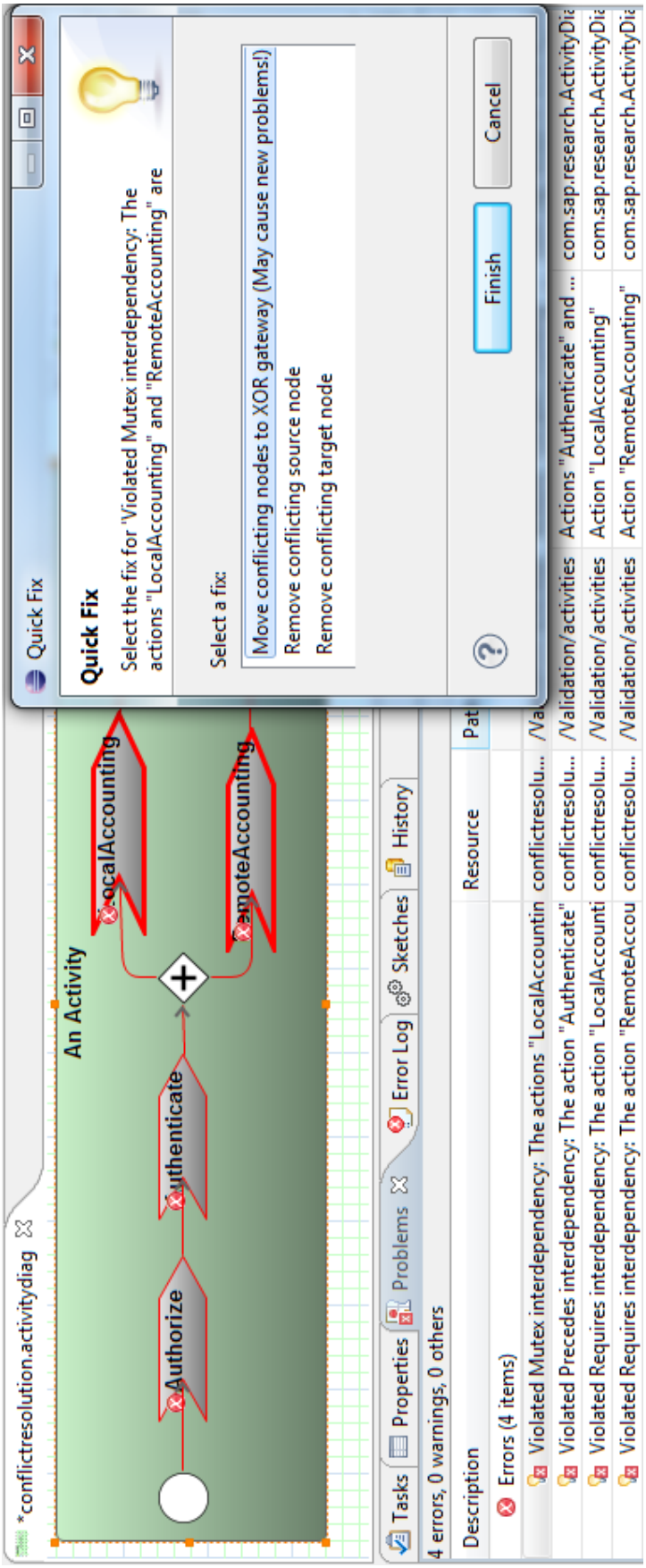

Figure 3.19: The Composition Editor: Conflict Detection and Resolution

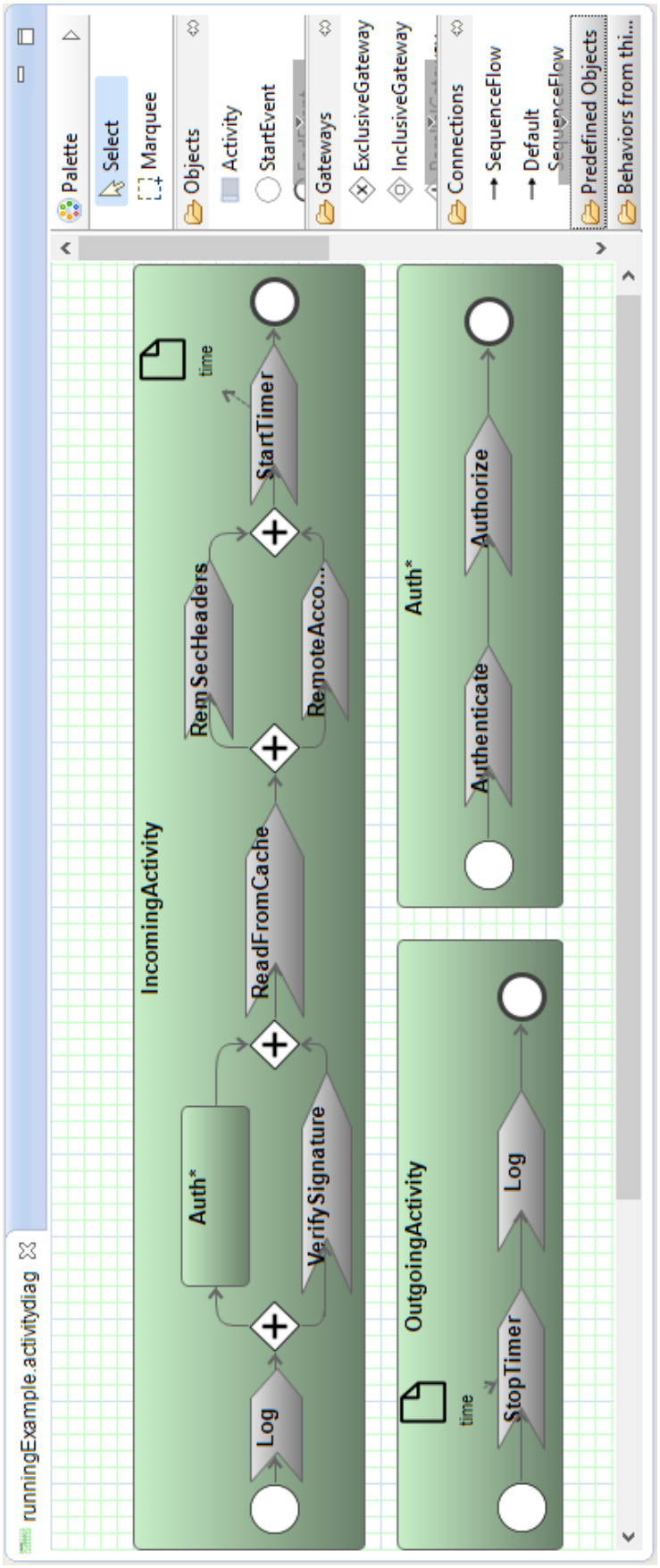

Figure 3.20: The Composition Editor: Resulting Non-Functional Activities

3.4.4 Action to Service Mapping

In the *Action to Service Mapping Phase*, the *Service Provider* chooses a set of web services she wants to enhance with NFAs. For web services regarded as black boxes the only available information is the service's WSDL interface. Hence, subjects such as particular operations, message types such as input, output or faults are suitable targets for NFAs. In the following, the semantics of the different subjects are explained. A *Service* subject has the same semantics as applying an NFA to all its messages independent of the type of message. In contrast, the *Input*, *Output* or *Fault* subject of a service applies NFAs only to the respective type of message. The same pattern is also available on operation level. An NFA can be mapped either directly to the operation, which means it applies to all kinds of messages consumed or produced by the operation or to individual types, for instance *Input*, *Output* or *Fault*. A detailed description of all supported black box subjects is given in Table 3.6.

In the black box view for web services, the abstract metamodel shown in Figure 3.11 is extended by concrete realizations of the *NonFunctionalAssociationTarget* interface. The resulting metamodel is shown in Figure 3.21. The *NonFunctionalAssociation* points to an interface called *NonFunctionalAssociationTarget* (reference to *NonFunctionalBehavior* omitted). This interface is realized by *ServiceRef* and *OperationRef* referencing the respective concepts in WSDL. The message directions can be controlled by the direction property of *NonFunctionalAssociation*. The subjects listed in Table 3.6 are represented by the combination of the direction type and the concrete association target. The *Service Request* subject, for example is represented by the *ServiceRef* class and the *direction* property set to *IN*.

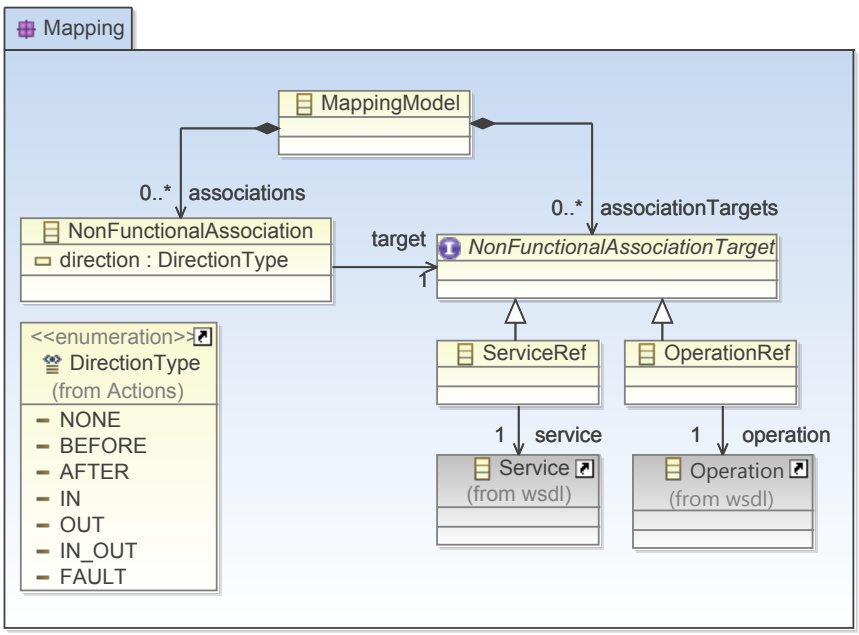

Figure 3.21: The Extended Mapping Metamodel for Services from the Black Box View

Subject	Description
Service Invocation	An action is executed before and after the invocation of any operation of a specific service. **Direction property**: *IN_OUT*. **Association Target**: *ServiceRef*. **Example**: All messages should be logged except faults.
Service Request	An action is executed before the request message is processed by a specific service. **Direction property**: *IN*. **Association Target**: *ServiceRef*. **Example**: An *Authorize* action checks whether the consumer is allowed to invoke the service.
Service Response	An action is executed after the response message has been produced by a specific service. **Direction property**: *OUT*. **Association Target**: *ServiceRef*. **Example**: An *Encrypt* action encrypts the message to make the transmitted data confidential.
Service Fault	An action is executed if a fault occurred during a service invocation. **Direction property**: *FAULT*. **Association Target**: *ServiceRef*. **Example**: A Compensation action catches the fault, logs it and invokes an alternative service.
Operation Invocation	An action is executed before and after the invocation of a specific operation of a service. **Direction property**: *IN_OUT*. **Association Target**: *OperationRef*. **Example**: Only idempotent operations should be cached.
Operation Request	An action is executed before the request message sent to a specific service operation arrives. **Direction property**: *IN*. **Association Target**: *OperationRef*. **Example**: A critical operation that receives banking data must be secured by an *Encrypt* action.
Operation Response	An action is executed before the response message replied by a specific service operation is sent. **Direction property**: *OUT*. **Association Target**: *OperationRef*. **Example**: A *Sign* action is executed only for a specific operation response that provides data that is critical to change by a malicious party.
Operation Fault	An action is executed if a (specific) fault occurred during a specific operation invocation. **Direction property**: *FAULT*. **Association Target**: *OperationRef*. **Example**: Faults that occur for a critical operation may be important so that a *Log* action sends an SMS to the Administrator.

Table 3.6: Subjects of NFAs in the Black Box View

3.4.4.1 Notation

Figure 3.22 shows the notation for the mapping to a service subject. The service is represented by a dark colored, rounded rectangle, while non-functional actions are represented by arrow shapes. The *direction* property of the *NonFunctionalAssociation* class are identified by different symbols decorating the connection line: *IN_OUT* — no symbol, *IN* — message symbol with an incoming arrow, *OUT* — message symbol with an outgoing arrow and *FAULT* — triangle symbol with a exclamation mark. In Figure 3.23 the notation for mapping to operation subjects is shown. Operations are represented by brightly colored, rounded rectangles.

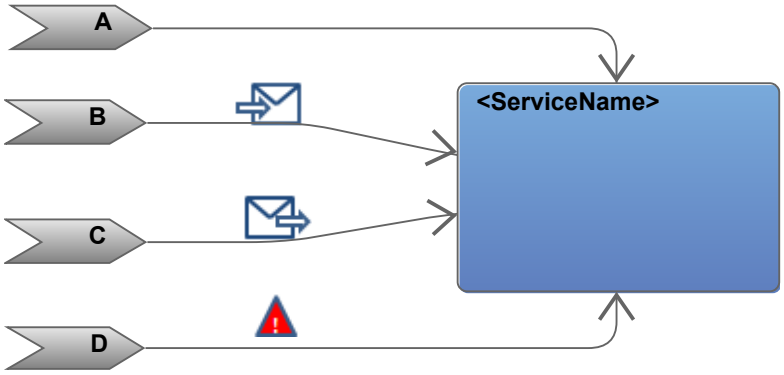

Figure 3.22: Mapping Notation for Services and Actions

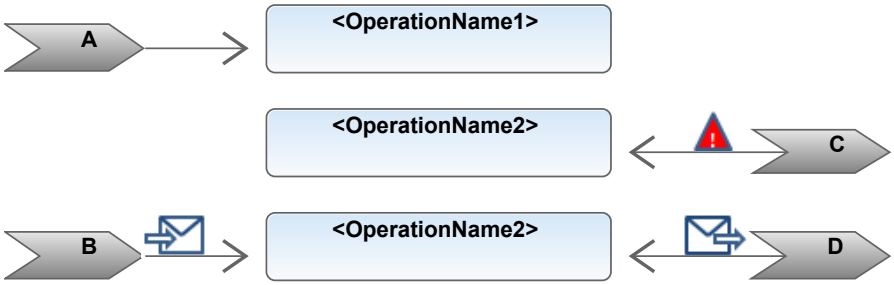

Figure 3.23: Mapping Notation for Operations and Actions

3.4.4.2 Running Example

The service provider maps the previously modeled non-functional actions and activities using the Mapping Editor. She decides to enhance her *FlightreservationWebService* with additional non-functional actions. Thus, she imports the WSDL of the web service which defines three operations, *searchFlights*, *searchBestFlight* and *bookFlight*. After importing the WSDL, the editor shows the service symbol (a rounded rectangle with the name of the service). Because the service provider would like to define a fine-grained mapping based on the service operations, she drills down to the operations view of the editor shown in Figure 3.24. The *searchFlights* and *searchBestFlight* operations do not change the state of the service, and customers must pay

for the use of these operations (e.g., travel agencies). Thus, she maps the *IncomingActivity* to incoming messages and the *OutgoingActivitiy* to outgoing ones to these operations. The *bookFlight* operation is state changing: A new flight will be booked. Thus, caching cannot be used. Moreover, the service provider decides that customers need not pay for the invocation of this operation, because she charges a 5% fee relative to the price of the flight. Consequently, accounting is also not required. Hence, she decides to use only authentication and authorization which has been modeled by the *Auth** activity which she maps for incoming messages to the *bookFlight* operation. Additionally, she wants to have logging for incoming and outgoing messages. This mapping leads to an undefined ordering between *Auth** and *Log* for incoming messages. However, the order is not important for the service provider, and thus the mapping fits her requirements.

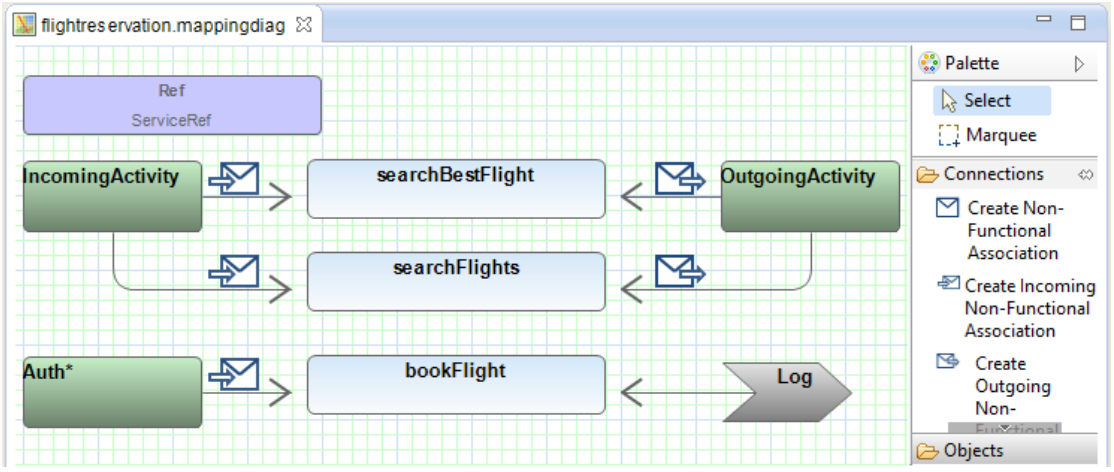

Figure 3.24: The Mapping Editor: Mapping NFAs to Web Services

3.4.5 Action to Middleware Service Mapping

In web services the components implementing non-functional behavior are often middleware services. Hence, the abstract Action Realization Mapping Phase is called *Action to Middleware Service Mapping* phase. In the *Action to Middleware Service Mapping* phase the *Domain Experts* select concrete middleware services that implement the specified NFAs. Each NFA can then be associated with a specific web service operation (if the middleware service is implemented as web service) that implements the action. This is reflected by the extended metamodel shown in Figure 3.25.

A *MiddlewareMapping* element has a reference to a WSDL operation implementing the action. Alternatively, if the middleware service is not implemented as a web service, the attribute *localName* can be used to point to a particular local software module implementing the action. For example, if the web service to be enhanced by additional NFAs runs on a platform which provides a set of local platform services, these can be referenced in the *localName* attribute. However, using this attribute makes the model platform dependent because the code generator

must interpret the value of the attribute for a specific platform (see Section 3.4.6). Nonetheless, it allows the realization of more efficient implementations of NFAs.

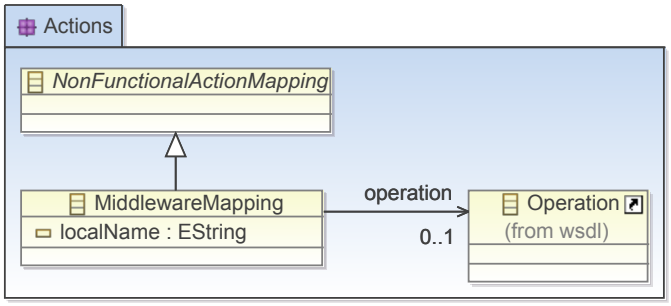

Figure 3.25: The Extended Action Metamodel for Middleware Mapping Support

Middleware services often provide several configuration options. For example, NFAs like *StrongEncrypt* and *WeakEncrypt* could map to the same security middleware service but different encryption algorithms. Sometimes the configuration depends on the service the NFA is mapped to (in the *Action To Service Mapping Phase*). Hence, this configuration can be defined not only on a per NFA basis but also per mapping between actions and services. Due to the extremely high number of possible actions and middleware services no standard configuration options can be provided. However, the configuration options can be specified by arbitrary key-value pairs. There are two types of configuration values: static and dynamic ones. Static values are set at design time. Dynamic values, however, can change, depending on the concrete execution context of the NFA. An NFA logging the message sent to or received by a web service can be considered an example. The logging NFA needs access to the context data, in this case the intercepted message. This is achieved by the use of context variables provided by NFComp. An overview on the supported variables available in the black box view is given in Table 3.7.

Variable	**Type**	**Description**
$message	XML	The intercepted message.
$header	XML	The header of the intercepted message.
$body	XML	The body of the intercepted message.
$serviceName	String	The name of the target web service.
$operationName	String	The name of the target operation.
$request	XML	The request message also available for actions mapped to *out* direction. For *in* direction $message equals $request.

Table 3.7: Context Variables Available in the Black Box View

Instead of a variable, expressions can also be used, for example, to query for a particular part of the variable. For the presented variables, XPath is an appropriate query language since the complex datatypes used here are all part of the SOAP XML Schema[3]. This allows the query of particular parameters in a message, for example, by simply writing

[3]http://www.w3.org/2003/05/soap-envelope/

$request/soap11:Body/<opname>/<paramname>/text() to get the parameter values of a particular operation.

Running Example

The *Performance Expert* configures the *ReadFromCache* action by mapping it to the existing *CachingWebService* which has already successfully been used by her company for other applications. Thus, she opens the Action Editor and selects the *ReadFromCache* action in the model. Then she imports the WSDL of the *CachingWebService*. The property sheet shows the available operations, and the table below shows the configuration entries that have been set to configure the middleware service. Figure 3.26 shows how the action editor can be used to accomplish the mapping. The modeler specified the *timeToLiveSeconds*, *overflowToDisk* and *maxLocalHeap* parameters. Additionally (not shown in the figure), she specifies the *message* parameter and sets its value to *$message* in order to use the intercepted message as key for the lookup in the cache.

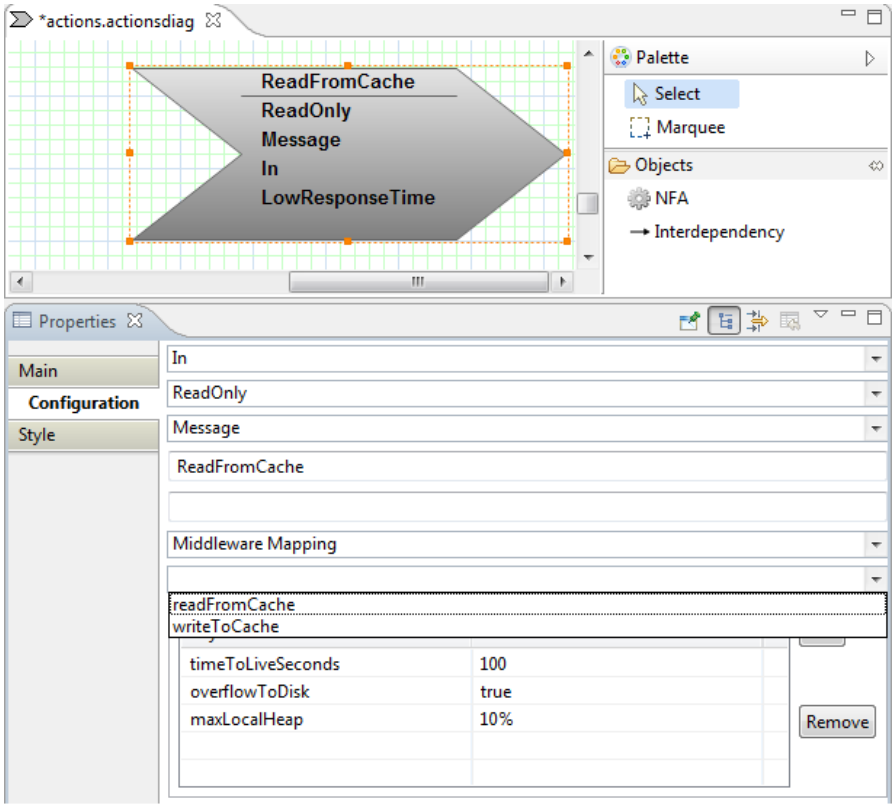

Figure 3.26: The Action Editor: Configuration of the Middleware Service Mapping

The *Accounting Expert* defines the mapping from the *RemoteAccounting* to the *AccountingWebService* and sets the configuration entries shown in Table 3.8. The most interesting configuration parameter is that of the *customerId*. It extracts the username from the WS-Security header of the soap message and passes it to the *AccountingWebService* which provides an operation *accountFor(customerId, marketplaceUrl, serviceId, operation, message)*. The *marketplaceUrl*

is an example for a static value and is used to determine the service marketplace, used to do the billing of the service usage.

Key	Value
customerId	$request/soap11:Header/wsse:Security/ wsse:UsernameToken/wsse:Username/text()
marketplaceUrl	http://marketplace.premium.de/remote
serviceId	$serviceName
operation	$operationName
message	$message

Table 3.8: Configuration Entries for the Accounting Middleware Service

3.4.6 Generation of NFC Enforcement Code

After the *Action to Service Mapping* and *Action to Middleware Service Mapping* phase the *Service Provider* generates the NFC enforcement code from the NFC specification model. The code generation is performed by dedicated code generator modules which generate proxy components or at least the configuration of an existing one.

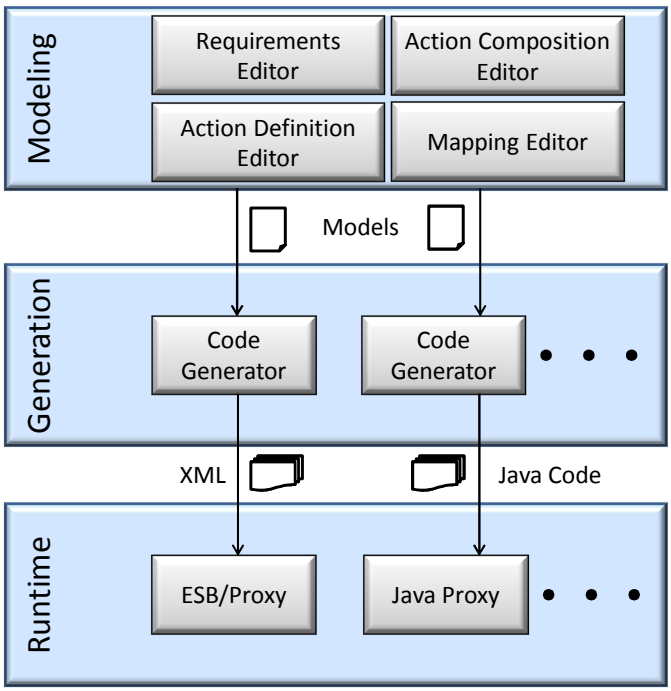

Figure 3.27: Generating Code Out of the Models

Different strategies can be used to provide proxies at runtime. Generally, the proxy intercepts web services calls and invokes the middleware services in the correct order according to the specification in the model. NFComp provides an implementation of a code generator for the

Apache Synapse[4] ESB (enterprise service bus). However, other code generators for other target platforms can also be developed. For example, instead of Apache Synapse, a Java program can be generated instead or any other ESB can be used. For each target platform, a new code generator must be implemented, but the same NFComp model can be used. Figure 3.27 depicts the process of code generation from a bird's eyes view.

3.4.6.1 Proxy-Based Enforcement of NFCs

In NFComp, a proxy configuration is generated for the Apache Synapse ESB. The *Service Provider* installs this proxy in front of her web services and changes the address location of the WSDL description to that of the proxy component. Consequently, whenever a potential service consumer invokes the web service she invokes the proxy instead (shown in Figure 3.28). The proxy executes the middleware services corresponding to the NFAs (as specified in the *Action to Middleware Service Mapping*) that have been mapped to the web service (as specified in the *Action to Service Mapping*). The task of executing middleware service is performed by so called mediators (depicted in Figure 3.28).

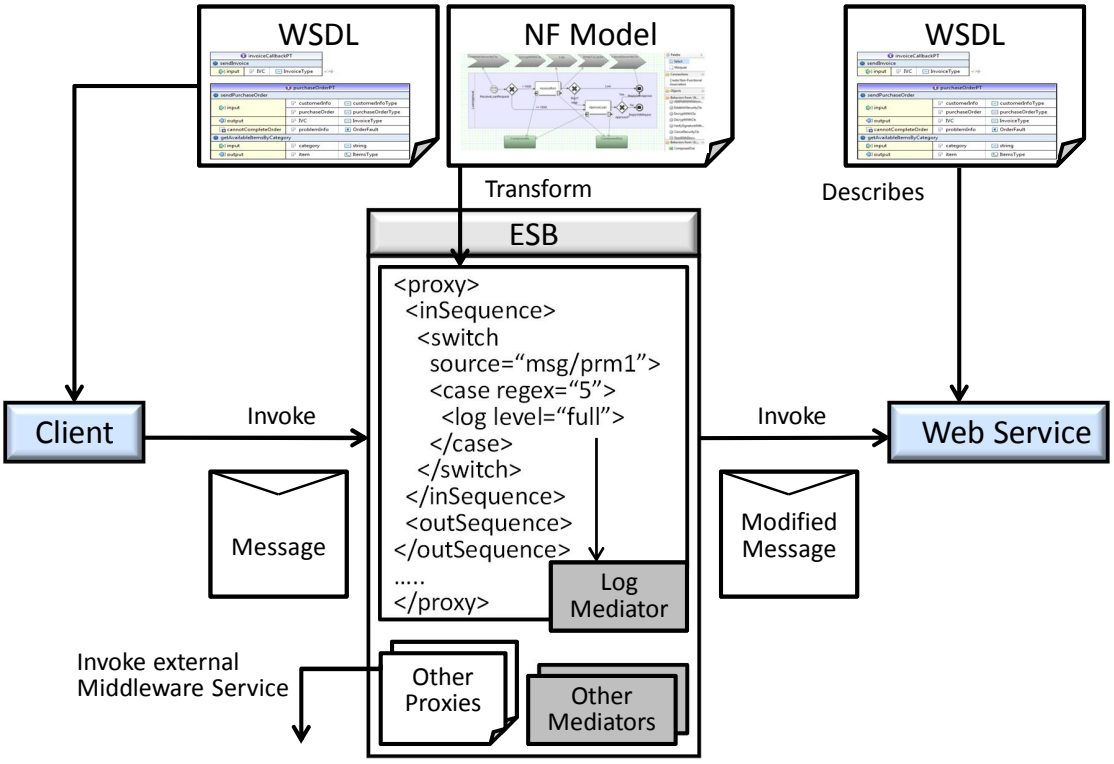

Figure 3.28: Runtime Enforcement of the Modeled NFCs

Finally, the proxy delegates the message to the target web service. After the web service produces the response, it will again pass the message to the proxy before it is delivered back to the service consumer. The advantage of this approach is that the proxy is decoupled from

[4] http://synapse.apache.org/

the programming platform of the intercepted web service, which is in contrast to the common handler approach (which is often used in SOAP frameworks, for example in Apache Axis2). This facilitates the reuse of the enforcement code for different web services independently of the programming language being used for their implementation.

Mediators The mediator concept plays an important role in the proxy-based NFC enforcement. A mediator is a message processing component in Apache Synapse. The input of a mediator is the intercepted input message which can be processed. Then the potentially modified message is handed over to the next mediator in the chain. The processing logic can be arbitrary Java code to be executed, such as adding or removing information from the message and performing data base lookups, among others. In Apache Synapse, there are different categories of mediators: core mediators, filter mediators, transformation mediators, extension mediators and advanced mediators. NFComp extends this set of available mediators by its own mediators: *ContextCollection* (*collectctx*), *RemoteMiddlewareService* (*callremotemwservice*), *Encrypt*, *Decrypt*, *LogData*, *Caching*, *Authenticate*, *AddAuthenticationData* (*addauthdata*), *StartTimer*, *StopTimer* and *Sign*.

A mediator in Synapse is a Java class implementing the *org.apache.synapse.Mediator* interface which defines among others the *mediate(org.apache.synapse.MessageContext ctx)* method. This method must be implemented by every mediator and contains the actual processing logic. Additionally, the method provides access to the message context. This message context, in turn, allows access to the intercepted message, configuration parameters, and so on. Synapse offers two ways to add a new custom mediator: using a generic mediator which points to a Java class (implementing a particular interface) or extending the XML configuration language by new XML elements, for example, new mediators. The second way is more sophisticated and has been chosen for NFComp.

3.4.6.2 Proxy Generation

The generation process is accomplished by different components which are orchestrated by a model workflow engine file (MWE[5]).

The mapping model file (which is an XMI file) is read by the *ModelReader* component, which turns the input into an in-memory Ecore model. In this process, the other referenced models (such as action and activity model) are loaded on-demand. To turn the XMI into an Ecore model, a set of metamodels (listed in Table 3.9) is used for transforming the input models into the Ecore representation. Some of those metamodels are based on EMF, i.e., there are Java (Ecore) classes implementing this metamodel (used whenever available, for instance offered by Eclipse plug-ins) and some are XSD-based, i.e., they use an automatic transformation from XML into a dynamic Ecore representation.

[5] `http://www.eclipse.org/modeling/emft/?project=mwe`

Metamodel	Type
SOAP	EMFMetaModel
WSDL	EMFMetaModel
HTTP	EMFMetaModel
MIME	EMFMetaModel
NFComp	EMFMetaModel
Synapse	XSDMetaModel
XSD	XSDMetaModel

Table 3.9: Metamodels Used for Code Generation

Then, the model is validated against particular rules (such as, for example, that associations must have a source and target or, that each action must have a name). If this validation succeeds, the transformation components will take the input Ecore model and create a model instance of the target model (which is the XML file conforming to the Synapse XML Schema). For this purpose, a set of Xtend[6] templates is used which define the rules for the transformation. XTend is a functional programming language which has been designed with a focus on model transformations.

The transformation is split into different components. The *ProxyTrafo* component creates the proxy configuration files, one per target web service. A proxy configuration defines the target location of the proxied web service and defines when which mediator sequence must be invoked. The *ActionTrafo* creates a set of mediator sequences, more specifically, one sequence file per action or non-functional activity. The created sequence will have the same name as the action, respectively activity. The *AssociationTrafo* Component creates one mediator sequence per non-functional association which points to one of the mediator sequences produced by the *ActionTrafo* component. This assures the reuse of generated configurations. Finally, the *WSDLTrafo* generates a local configuration entry for Synapse in order to find the WSDL of the target, proxied web service. For each transformer component, there is one *XMLWriter* to write the target XML files to the hard disk. Figure 3.29 depicts the generation process with the different components.

The generated code for the Apache Synapse ESB is written into different output folders: *proxy-services*, *local-entries* and *sequences*. The *proxy-services* folder contains one proxy configuration per proxied web service (see Proxy XML in Figure 3.29). The *local-entries* folder contains local configuration entries for XML Schemas, WSDL files (see *WSDLEntry* XML in Figure 3.29) or other resources. Finally, the *sequences* folder contains mediation sequences (*ActSeq* XML and *AsSeq* XML), i.e., a list of mediators performing a predefined action.

[6] `http://www.eclipse.org/xtend`

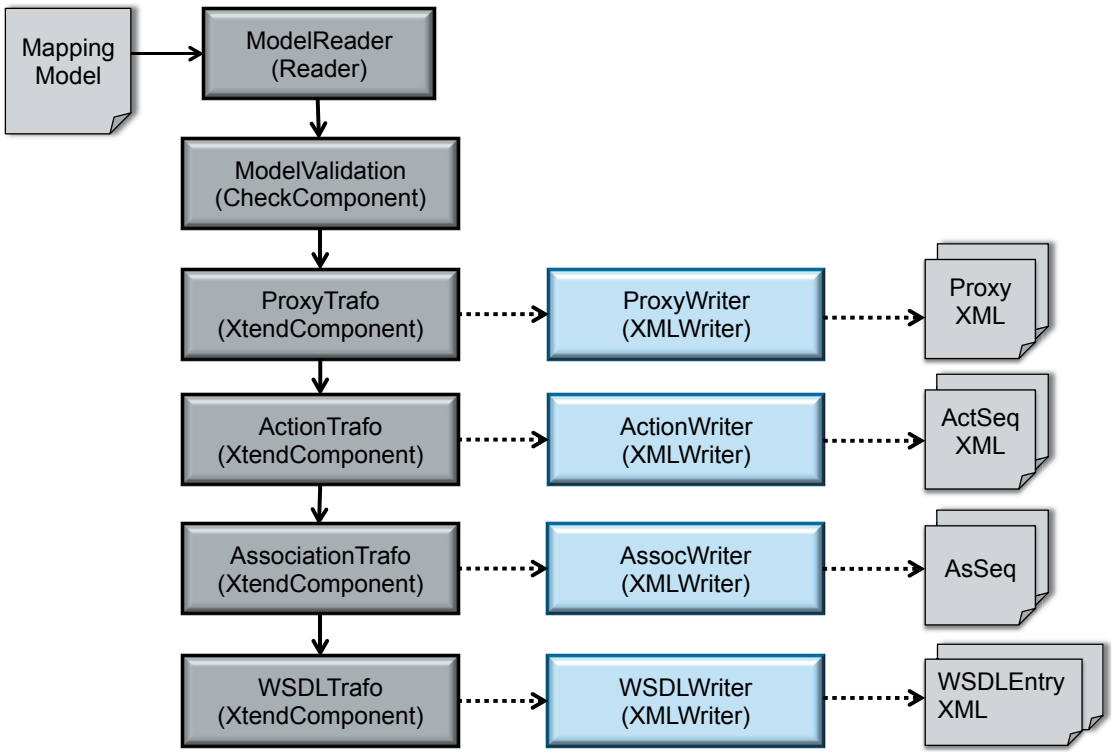

Figure 3.29: Generator Components

3.4.6.3 Support for Web Service Consumers

As discussed in Section 2.2.5 there are particular NFCs which require the consideration of several parties and not only one component in isolation. To address this requirement NFComp does not only support the service provider, but also the consumer of a web service to compose non-functional concerns. The whole approach can be applied to the consumer side. To accomplish this, the Apache Synapse ESB must also be installed at this side. Then, the target web service URL must be replaced by the one of the proxy, which then delegates the message to the actual web service after invoking additional middleware services.

The use of middleware services at the consumer side is especially important to stay interoperable with the service, e.g., when it makes use of encryption, signatures or reliable messaging. In this case the consumer should be able to decrypt and verify signatures or to understand the reliable messaging protocol. It is likely that both consumer and provider will make use of NFComp. For this case, NFComp provides a mechanism for the consumer to determine the action composition automatically from the service's action composition. This works as follows: The service consumer takes the service's NFComp model and configures the code generator to produce the inverse logic. This is achieved by setting the *consumerworkflow* property of the MWE workflow to *true*. This property will then be passed to the *ProxyTrafo*, *ActionTrafo* and *AssociationTrafo* component which will generate the inverse order for all non-functional activities mapped to the service.

The inverse order is calculated with help of the *inverse* interdependency. Each task executing a particular non-functional action, is replaced by a task calling its inverse action. If there is no such inverse action, the task will be removed from the process. Then, the direction of all sequence flows is inverted and start and end events are interchanged. Finally, the expressions of conditional sequence flows are moved to the sequence flow connections leaving the opposite gateway. This mechanism allows the generation of two proxy configurations out of one and the same model: one for the consumer and one for the provider of the service. This feature has also been included in Eclipse, where the user can configure the destination folder to which the proxy configuration should be written. The provider can thus generate the configuration to a consumer version of the ESB and offer this preconfigured proxy as download for potential consumers. The whole approach could, instead, also be applied at the modeling level. The consumer could generate a new model compatible to the providers NFComp model. This feature, however, has not yet been implemented.

3.4.6.4 Running Example

The *Service Provider* takes the NFC specification for her flight reservation service that is already running on a dedicated application server. She creates the ESB configuration by generating a set of XML configuration files. The ESB is configured to maintain a proxy for the flight reservation service with the two activities *IngoingActivity* and *OutgoingActivity*, respectively *Auth** and the *Log* action.

More specifically, the code generator produces the *MyFlightReservation_proxy.xml* with the configuration shown in Listing 3.1. The *target* tag (Line 2-34) defines which mediators will be applied to the target web service. The mediators can be defined in an incoming (*inSequence*, Line 3-19) and outgoing sequence (*outSequence*, Line 20-33) representing the incoming, respectively outgoing, message flow. The *switch* mediator (Line 4-15) used in the *inSequence* checks the *source* expression against the *regex* value in the different *case* tags. If one of the case *regex* values matches the result of the expression, the mediators defined for the respective *case* is executed. For this example, the mapping has been defined on an operation basis. Thus, the different operation names are checked by the *switch* mediator. If the operation name equals one of *searchBestFlight* (Line 5), *searchFlights* (Line 8) or *bookFlight* (Line 11), the corresponding association mediator sequences are invoked. These sequences are defined in separate files. The *key* attribute matches the name of the sequence which is a composition of the operation name and the activity or action name, e.g., *searchBestFlight_IncomingActivity* for the *searchBestFlight* operation.

After the *switch* mediator, the *send* mediator (Line 16-18) is invoked. This mediator is responsible for finally delivering the message to the target web service. Then, after execution of the target web service, the response message will be received by the ESB and the *outSequence* is started. Again, depending on the operation, different mediator sequences are invoked.

```
1  <proxy xmlns="http :// ws.apache.org/ns/synapse"  name="MyFlightReservationService"  transports ="http , https ">
2    <target >
3      <inSequence>
4        <switch  source="get-property('OperationName')">
5          <case  regex=" searchBestFlight ">
6            <sequence key=" searchBestFlight_IncomingActivity "/>
7          </case>
8          <case  regex=" searchFlights ">
9            <sequence key=" searchFlights_IncomingActivity "/>
10         </case>
11         <case  regex="bookFlight">
12           <sequence key="bookFlight_Log"/>
13           <sequence key="bookFlight_AuthStar"/>
14         </case>
15       </switch>
16       <send>
17        <endpoint><address  uri="http://localhost:7080/../MyFlightReservationService"/></endpoint>
18       </send>
19     </inSequence>
20     <outSequence>
21       <switch  source="get-property('OperationName')">
22         <case  regex=" searchFlights ">
23           <sequence key=" searchFlights_OutgoingActivity "/>
24         </case>
25         <case  regex=" searchBestFlight ">
26           <sequence key=" searchBestFlight_OutgoingActivity "/>
27         </case>
28         <case  regex="bookFlight">
29           <sequence key="bookFlight_Log"/>
30         </case>
31       </switch>
32       <send/>
33     </outSequence>
34   </ target >
35   <publishWSDL key="MyFlightReservationService_wsdl"/>
36 </proxy>
```

Listing 3.1: Proxy Configuration for *MyFlightReservationService*

In addition to the proxy configuration file, the mentioned mediator sequence files are generated. For each modeled non-functional association, there is one such sequence file. Listing 3.2 shows the association sequence between the service operation *searchFlights* and the non-functional activity *IncomingActivity*. The sequence *searchFlights_IncomingActivity* references another sequence (Line 2) representing the *IncomingActivity*.

```
1 <sequence xmlns="http :// ws.apache.org/ns/synapse"  name="searchFlights_IncomingActivity">
2   <sequence key="IncomingActivity"/>
3 </sequence>
```

Listing 3.2: Association Sequence Between *searchFlights* and *IncomingActivity*

Finally, the sequences containing the mediators which are responsible for the non-functional action realization are generated. There are generally two types of mediators: standard Synapse

mediators and custom mediator extensions. NFComp uses existing Synapse mediators whenever possible. However, not every middleware feature provided by Synapse can be used in such a flexible and fine-grained way as required for NFComp. For example, for security there is a *<enableSec [policy="key"]/>* element, and for reliable messaging there is a *<enableRM [policy="key"]/>* element in order to activate security and reliable messaging. Both can be configured via WS-Policy documents. It is however not possible to separate the process of signing and encryption which is required, among others, for this running example. Furthermore, particular non-functional features are not supported at all; for example, for the *StartTimer* and *StopTimer* actions there is no adequate mediator.

In the following, the mediators and their configuration are explained. Notice that Synapse does not yet support parallel execution of mediators. Thus, the parallelism is transferred into sequential execution of mediators. The *log* mediator[7] logs the intercepted SOAP message. It has a configurable log level which can be one of *simple|full|headers|custom*. Simple is the default level logging *To*, *From*, *WSAction*, *SOAPAction*, *ReplyTo* and *MessageID* headers. *Full* additionally logs the message payload, *headers* logs all SOAP headers and *custom* logs user defined properties. The *sign* mediator is a custom mediator, adding a signature to the SOAP header if it is a response message, or verifying the signature if it is a request message. The *callremotemwservice* mediator is an extension to the standard Synapse *callout* mediator, allowing the addition of arbitrary properties for SOAP operation input parameters. It has three mandatory parameters *config_operation*, *config_namespace*, *config_serviceUrl* for configuring the web service which is consumed. *config_output* defines in which variable the response of the consumed middleware service is to be stored. All other parameters are used as input parameters for the SOAP message. This mediator is used for all actions being mapped to middleware web services. All other mediators are generated using a name mapping (the *localName* attribute in the middleware mapping) to the corresponding Synapse mediator.

Listing 3.3 shows the generated *IncomingActivity* as an example. The first *callremotemwservice* mediator (Line 9-17) calls the *AccessControlService* with the parameters *service* mapped to the *$serviceId* variable (Line 15), *operationName* mapped to the *$operationName* parameter (Line 16) and *userId* mapped to an expression extracting the username token from the message header (Line 14). The next *callremotemwservice* mediator (Line 18-28) consumes the *CachingWebService* according to the *ReadFromCache* action. The *header* mediator (Line 29) is a standard Synapse mediator manipulating headers of the intercepted SOAP message. It can be configured via the action attribute to *set* the header to a value or to *remove* the header. In this case it realizes the *RemSecHeaders* action and thus removes the security header. The remote accounting action is realized by another *callremotemwservice* mediator (Line 30-40) configured to pass the necessary parameters to the *AccountingWebService*. Finally, the *StartTimer* action is realized as a custom *startTimer* mediator (Line 41-43), capturing the current system time and writing it to a variable stored in the local registry of Synapse.

[7] `http://synapse.apache.org/userguide/mediators.html#Log`

The name of the variable in the registry is defined by the *producedVariableKey* property, in this case *timer* (which is the name of the data item modeled in the action composition diagram).

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="IncomingActivity"
    onError="fault">
  <log level="full"/>
  <sign/>
  <authenticate>
    <property name="acceptedUserName" value="testuser"/>
    <property name="acceptedPassword" value="password"/>
  </authenticate>
  <callremotemwservice>
    <property name="config_output" value="Authorize"/>
    <property name="config_operation" value="checkAccess"/>
    <property name="config_namespace" value="http://AccessControl.ws.bs.de"/>
    <property name="config_serviceUrl" value="http://localhost:7080/axis2/services/AccessControlService.
        AccessControlServiceHttpSoap11Endpoint/"/>
  <property name="userId" value="$request/soap11:Header/wsse:Security/wsse:UsernameToken/wsse:Username/text()"/>
    <property name="service" value="$serviceId"/>
    <property name="operationName" value="$operationName"/>
  </callremotemwservice>
  <callremotemwservice>
    <property name="config_output" value="ReadFromCache"/>
    <property name="config_operation" value="readFromCache"/>
    <property name="config_namespace" value="http://cachingmws.ws.bs.de"/>
    <property name="config_outOnly" value="false"/>
    <property name="config_serviceUrl" value="http://localhost:7080/axis2/services/CachingService.
        CachingServiceHttpSoap11Endpoint/"/>
    <property name="timeToLiveSeconds" value="100"/>
    <property name="overflowToDisk" value="true"/>
    <property name="maxLocalHeap" value="10%"/>
    <property name="message" value="$message"/>
  </callremotemwservice>
  <header action="remove" name="wsse:Security"/>
  <callremotemwservice>
    <property name="config_operation" value="createAccountingDatum"/>
    <property name="config_namespace" value="http://accounting.ws.bs.de"/>
    <property name="config_outOnly" value="true"/>
    <property name="config_serviceUrl" value="http://localhost:7080/axis2/services/AccountingService.
        AccountingServiceHttpSoap11Endpoint/"/>
    <property name="customerId" value="$request/soap11:Header/wsse:Security/wsse:UsernameToken/wsse:Username/
        text()"/>
    <property name="marketplaceUrl" value="http://marketplace.premium.de/remote"/>
    <property name="serviceId" value="$serviceId"/>
    <property name="operation" value="$operationName"/>
    <property name="message" value="$message"/>
  </callremotemwservice>
  <startTimer>
    <property name="producedVariableKey" value="time"/>
  </startTimer>
</sequence>
```

Listing 3.3: Activity Sequence for IncomingActivity

3.5 NFC Composition in a Gray Box View of Web Services

In this section, NFComp is presented from the gray box view. Gray box in this regard means that — besides its interface — the composition logic of a composite service is also visible. Compared to the black box view, more assumptions must be made on one hand, for instance that the service is composite, but on the other hand there also new possibilities with respect to non-functional concerns.

In this section, the different phases are revisited and the extensions regarding the gray box view are described. Especially for the three phases *Action Composition*, *Action to Composite Service Mapping* and *Generation of NFC Enforcement Code* there are remarkable new concepts which merit consideration. In the *Action Composition Phase* there is, as with service, a distinction between atomic and composite NFAs. In the *Composite Service Mapping Phase*, there are new mapping targets such as the process logic of the composite service. Finally, in the *Generation of NFC Enforcement Code Phase*, the proxy-based approach must be adapted in such a way that the proxy is aware of the internal process state, e.g., which activity of the process is currently executed.

3.5.1 Requirements Specification

In this phase, the *Requirements Engineer* compares the requirements regarding the composite web service with those of its partner services. Partner services, in this context, are services consumed by the composite service in order to fulfill its business goal. If a partner service requires confidentiality for particular messages, the process should assure confidentiality for all those messages sent to and received by this partner service. In this case, the list of the non-functional requirements to be supported will be extended by non-functional requirements demanded by partner web services.

On the other hand, requirements pertaining to the process may have an impact on the partner services. For example, if the process requires a good performance, then the partner services also should perform well to achieve this goal. If the partner services are managed using NFComp, the performance requirement, or generally the requirement of the process, can be forwarded to the list of requirements for the partner services. However, this is only a realistic option when the partner services are not provided by external parties. If this is not the case and NFComp cannot be applied, the poorly performing partner service can alternatively be exchanged for another service which performs better. Although the gray box view requires the consideration of both process and partner service requirements, no specific extension to the requirements model is necessary.

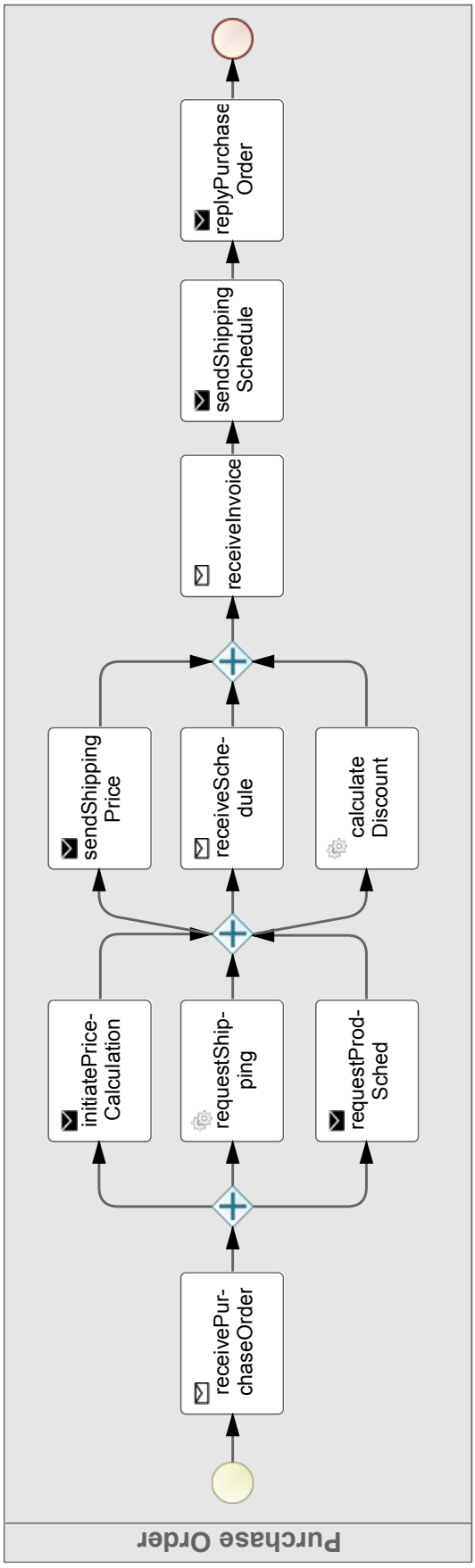

Figure 3.30: The Purchase Order Process

Running Example

The purchase order process defines a process for ordering goods via a web service interface. It is a composite web service consuming a set of partner web services. The control flow of the purchase order service has been defined as a BPMN2 process, which is shown in Figure 3.30. Furthermore, it is assumed that the purchase order process has been modeled in a purely functional way; i.e., it has been developed without support for non-functional concerns. However, there are different non-functional requirements to be supported in order to expose the service to real customers.

Generally, the purchase order process must perform well (low response time) and thus the partner services *ShippingService*, *CalculationService*, *SchedulingService*, *DiscountService* and *InvoiceService* need to perform well also. Since *DiscountService* is a third-party service and must be consumed over the Internet, it is the most critical one regarding performance. In summary, the following requirements have been identified:

1. The purchase order operation can only be called by authorized consumers (Security).
2. The consumption of the *DiscountService* which is used to calculate discounts based on the item quantity is called frequently and must perform well, i.e., the service must be consumed with a low response time (Performance).
3. The execution time of the process should be tracked in order to monitor any critical peaks (Monitoring).

3.5.2 Action Definition

In Section 2.2.3 different types of composition have already been identified. In the black box approach, the composition of actions to services and the vertical composition (non-functional activities) of actions play a dominant role. However, in the gray box view, not only services are considered composite but also actions themselves. Thus, there are two types of non-functional actions to deal with: atomic ones and composite ones. Atomic actions are defined as in the black box view. Composite actions are composed by atomic actions and are defined in the *Action Composition Phase*. Thus, there is no significant change to actions themselves, but there is a change regarding the scope of interdependencies between actions.

3.5.2.1 Scopes

NFComp defines an interdependencies concept between actions. Interdependencies can be of different types introducing different constraints (see Section 3.3.1). However, specifying the type of an interdependency is sometimes not sufficient. When considering composite services as a target for the mapping of actions, the extent to which a constraint regarding the actions is valid may not comprise the whole process execution but be restricted to limited *ranges* of the execution. Ranges for which a constraint is valid are defined as *scope* of the constraint. To

see that constraints must in general be defined with respect to a certain scope, the following real-world analogy can be considered: Regarding two house numbers, there is the restriction that they may not be equal; however, this is only valid if both are in the same street. Hence, this constraint is not valid in general ("general scope") but only in "street scope". Whenever constraints are used, their definition must define the scope in which the constraint shall be valid. This is similar to processes consisting of tasks: There may be two actions which exclude each other — but only for a single task — whereas there is no problem if they are both executed by different tasks in the process.

There are three distinct scopes restricting the validity of an interdependency to a certain level. Constraints imposed by certain interdependency types must be satisfied in exactly this scope ignoring actions outside this scope. The *interaction scope* restricts a constraint to a single interaction such as a request OR response. The *invocation scope* restricts a constraint to the invocation of a single operation: request AND response. The *process scope* defines that constraints for actions are valid for the whole process.

The *Interdependency* concept in the Actions Metamodel has thus been extended by a new attribute scope of type *ScopeType*. The *ScopeType* is an *EEnumeration* defining three *ELiterals* *INTERACTION*, *INVOCATION* and *PROCESS*.

3.5.2.2 Running Example

A set of non-functional actions must be identified in order to satisfy the previously defined requirements. To address Requirement 1, an *Authorize* action can be used that checks if a certain user has access rights to call the purchase order service. However, this *Authorize* action requires an *Authenticate* action in order to verify that the provided user ID belongs to the user who sent the message. This can be modeled as an interdependency of type *requires* from *Authorize* to *Authenticate* and an interdependency of type *precedes* from *Authenticate* to *Authorize*. To support Requirement 2, two actions *WriteToCache* and *ReadFromCache* can be used to write received responses to the cache in order to quickly retrieve the discount for previously ordered products from this cache. In this case, the *ReadFromCache* action must be defined before *WriteToCache*, and *ReadFromCache* requires *WriteToCache*. Requirement 3 can be solved by adding monitoring actions to the process, for instance a *StartTimer* and a *StopTimer* action. The *StopTimer* action requires the *StartTimer* action, and *StartTimer* precedes *StopTimer*.

These examples have in common that there are different actions with *precedes* and *requires* interdependencies. However, the scope of the interdependencies is on different levels. *Authorize* should be invoked for each interaction with an operation (request and response are both considered distinct interactions) of a web service with the constraint that, due to the *requires* interdependency, *Authenticate* also must be invoked for each such interaction. In case of *ReadFromCache* and *WriteToCache*, this implication would be too restrictive. *ReadFromCache* requires *WriteToCache* not for the same interaction (a single request OR response), but rather for the same invocation (a request-response instance). This means whenever *ReadFromCache*

is used for a service request, *WriteToCache* also must be used, namely for the corresponding response. In the case of the monitoring examples, this would again be too restrictive. In this case, the *StopTimer* requires the execution of the *StartTimer* action, not necessarily for the same interaction or invocation, but rather on the process level.

3.5.3 Action Composition

In the *Action Composition Phase*, two types of compositions must be defined: vertical and horizontal ones. Vertical compositions define the execution order and control flow of actions when applied to one and the same functional execution point. This type of composition has already been addressed in the black box view (Section 3.4.3). Horizontal compositions, however, define the execution order of actions executed at different functional points. This is required if a set of actions must be executed to realize a composite action. Examples for such composite actions are transactions, secure conversations, monitoring, among others. They require different non-functional actions to be executed in a well-defined order at different functional points, e.g., different points of the underlying functional process. Figure 3.31 shows the extended metamodel with the *CompositeNonFunctionalAction* which is a specialization of *NonFunctionalBehavior*.

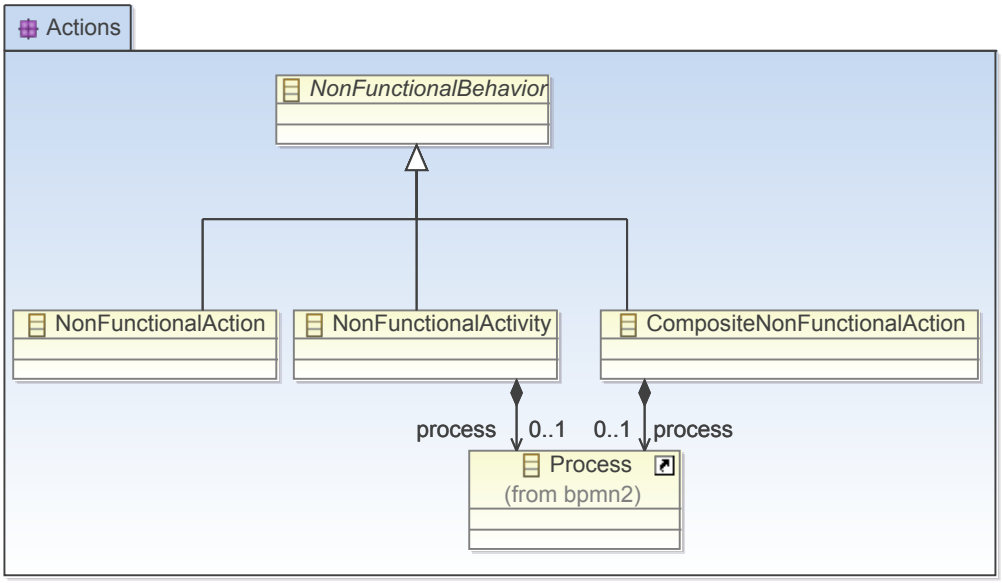

Figure 3.31: The Extended Metamodel for Action Composition in the Gray Box View

An advantage of composite non-functional actions over non-functional activities is that they are defined for exactly one non-functional concern and thus can be modeled by a single non-functional domain expert. In contrast, non-functional activities crosscut different non-functional domains such as performance, security or reliability and are modeled by the *Service Provider*. This was also the motivation for the definition of interdependencies between actions which can be validated for non-functional activities. However, it is more complex to assure that the actions defined in composite actions are mapped in the right order. For example, there could be two

tasks T1 and T2, and T1 is executed before T2. Furthermore, Action A1 and A2 may be part of a composite action A which prescribes that A1 must be executed before A2. Then, a mapping from A2 to T1 and A1 to T2 would contradict the composition logic of A. The order of actions in composite actions can — as in non-functional activities — be described with BPMN2, however, interdependencies can also be used. The advantage of using interdependencies is that this allows leveraging the validation logic already presented in Section 3.3. To combine the definition of composite actions in BPMN2 with the validation via interdependencies, NFComp defines a mapping from BPMN2 to interdependencies which is described in the following.

3.5.3.1 Deducing Interdependencies from Composite Actions

```
1  Let a and b bet tasks and a != b, then the following rules are applicable:
2  Rule 1: If there is a sequence flow from a to b
3    then precedes(a, b), requires(b, a), requires(a, b)
4  Rule 2: If x is a closing gateway or an opening parallel gateway and there is a sequence flow from a to x and
         x to b
5    then precedes(a, b), requires(b, a), requires(a, b)
6  Rule 3: If x is an opening exclusive gateway and there is a sequence flow from a to x and x to b
7    Case A: there is another exclusive branch besides the one containing b and this branch points into the same
           direction
8      then precedes(a, b), requires(b, a)
9    Case B: else
10     then precedes(a, b), requires(b, a), requires(a, b)
11 Rule 4: If a and b are part of two exclusive branches into the same direction of an XOR gateway pair and each
         action is executed by at most one task
12   then mutex(a,b)
```

Listing 3.4: Algorithm for Transforming Composite Actions into Interdependencies

The interdependencies derived from the composite action are all in *process scope*. What this means in terms of validation is elaborated in Section 3.5.4. The mapping from process constructs to interdependencies is given in Listing 3.4. The presented mapping for opening — respectively closing gateways exemplarily assumes that there are two successor, respectively two predecessor, nodes from the given gateway. However, the same mapping can also be applied accordingly for more than two successors/predecessors.

The generated interdependencies represent safe constraints for validation; i.e., they must be fulfilled according to the given composite action. However, there may be other interdependencies which cannot be deduced from the composite action process. Those interdependencies must be modeled manually in the *Action Definition Phase*.

3.5.3.2 Running Example

The actions *StartTimer* and *StopTimer* which have already been modeled in the *Action Definition Phase* can also be modeled as a composite action. Thus, a new composite action *Monitoring* must be created which is depicted in Figure 3.32. This action contains two atomic actions connected via a sequence flow. According to Listing 3.4, Rule 1 can be applied to the *Monitoring*

action resulting in two interdependencies *precedes(StartTimer, StopTimer)*, *requires(StopTimer, StartTimer)* and *requires(StartTimer, StopTimer)*. These are exactly those interdependencies that have already been modeled in the *Action Definition Phase*. However, modeling composite actions provides a new, more natural perspective on several atomic actions. It allows to focus on the process logic enabling a process-oriented view on non-functional concerns. The modeler should express as much as possible with composite actions because it is more straightforward and thus more efficient than using pure interdependency modeling. However, if there are interdependencies which cannot be expressed by composite actions, they must be modeled separately. The disadvantage, in this case, is that the information is distributed among the action and composition model and thus it should be considered to model all interdependencies directly.

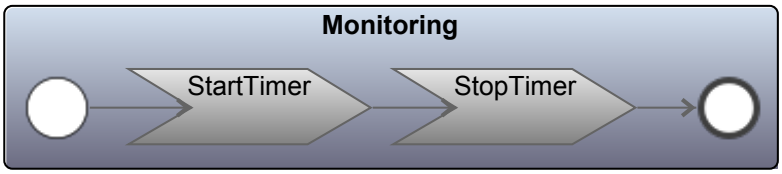

Figure 3.32: The Monitoring Composite Action

3.5.4 Action to Composite Service Mapping

In the *Action to Composite Service Mapping Phase*, the actions and activities defined in the previous phase are mapped to composite services, i.e. services exposing the composition logic explicitly by a particular kind of workflow language. There are two possible candidates for expressing the composition logic. One is WS-BPEL as executable XML-based web service standard. The other is BPMN2 as a graphical notation standard for business processes also executable and mappable to web services. Since the NFComp approach is on the graphical modeling level, it makes more sense to adhere to a graphical standard than to an XML-based notation. Thus, composite web services in NFComp must be defined in BPMN2 (a WS-BPEL mapping would also be possible, although a new kind of graphical representation for WS-BPEL would need to be defined in this case). In composite services new subjects for NFAs can be identified. All kinds of events which have a corresponding notation in the BPMN diagram are candidates for this: the execution of an activity, start or end events, errors, compensations among others. The gray box subjects supported by NFComp are summarized in Table 3.10.

Generally, as in the black box view, an NFA can be directly mapped with a non-functional association connection line pointing to tasks instead of operations or services. The association types introduced for the black box view can be reused in this context. Additionally, there are two additional *DirectionType* literals *Before* and *After*, which are, in contrast to *In*, *Out* and *In_Out* not targeting messaging events in the context of a task, but rather the execution of a task itself.

Figure 3.33 shows the extension of the mapping metamodel with the concepts of the *MessagingActivityRef* and the concrete classes *ServiceTaskRef*, *SendTaskRef* and

ReceiveTaskRef pointing to the corresponding concepts in BPMN2.

Subject	Description
Before Task	An action is executed before the associated task has been executed. **Example**: Starting a timer to measure execution time of the task.
Service or Send Task Request	An action is executed after the associated task has produced a request message and before sending it to the partner service. **Example**: An *Encrypt* Action encrypts the request message before it is sent to the partner.
Service or Receive Task Response	An action is executed after the response message has been received by the associated task and before it is processing it. **Example**: A *Decrypt* action decrypts the response message before it is processed by the task.
After Task	An action is executed after the associated task has been executed. **Example**: Stopping the timer to calculate the execution time of the task.
Task Error	An action is executed after the associated task has caused an error. **Example**: A *Counter* action counts the number of errors.

Table 3.10: Subjects of NFAs in the Gray Box View

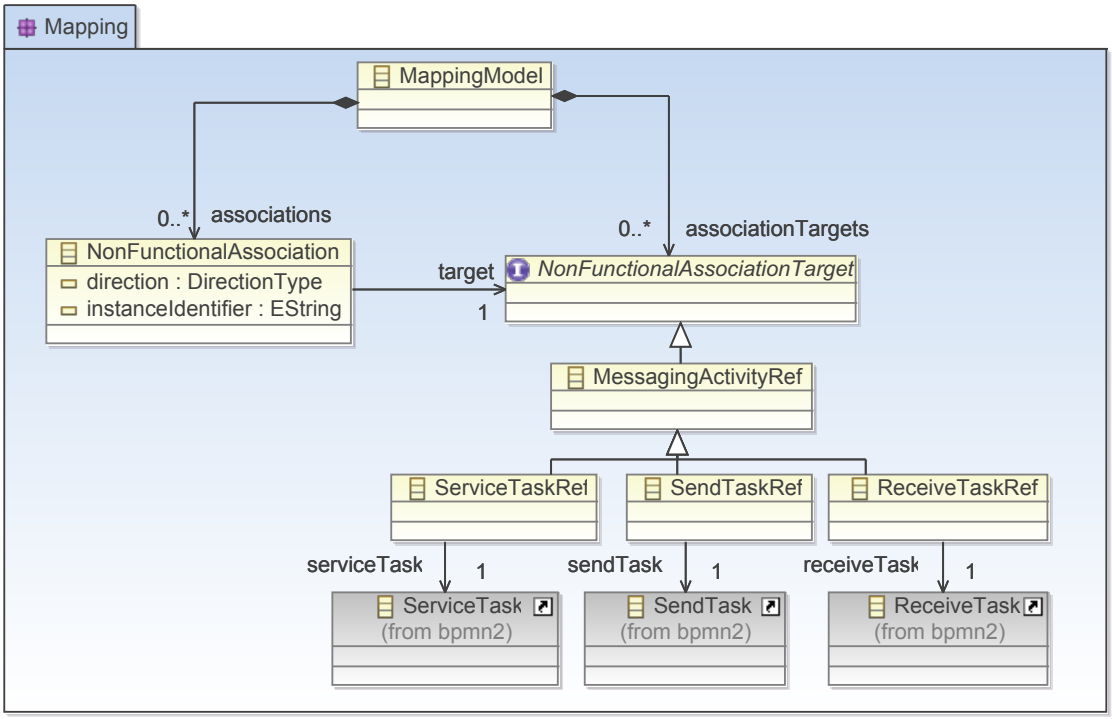

Figure 3.33: The Extended Metamodel for Action Composition in the Gray Box View

Furthermore, the *NonFunctionalAssociation* concept has been extended by a new attribute *instanceIdentifier*. This attribute is optional and can be used in the case when more than one

association is connected with one and the same action. Such actions are called multi-instance actions in the following. When a multi-instance action is used, it may be important to distinguish different instances of this action. When, for example, two actions *StartTimer* and *StopTimer* are mapped twice, and they pertain to the same composite action *Monitoring*, there are also two instances of of this composite action. The mapping of the *Monitoring* instance is hence unclear; it is not defined which *StartTimer* action and which *StopTimer* action belongs to which instance of *Monitoring*. In this case, the *instanceIdentifier* helps to distinguish the instances based on the non-functional associations pointing to the actions. The association to the first *StartTimer* action and the first *StopTimer* action could be set to A and the others to B to identify the two instances of *Monitoring*. The *instanceIdentifier* is important especially for composite actions managing state, for example, when they define data items.

Figure 3.34 depicts the mapping notation for the gray box view. Action A is mapped to *ReceiveTask* via *Before*, Action B is mapped to *ReceiveTask* via *In*, Action C is mapped to *ServiceTask* via *In_Out*, Action D is mapped to *SendTask* via *Out* and Action E is mapped to *SendTask* via *After*. The association types from left to right also reflect their execution order when being mapped to one and the same task, for instance *Before* is before *In*, *In* is before *In_Out* and so on.

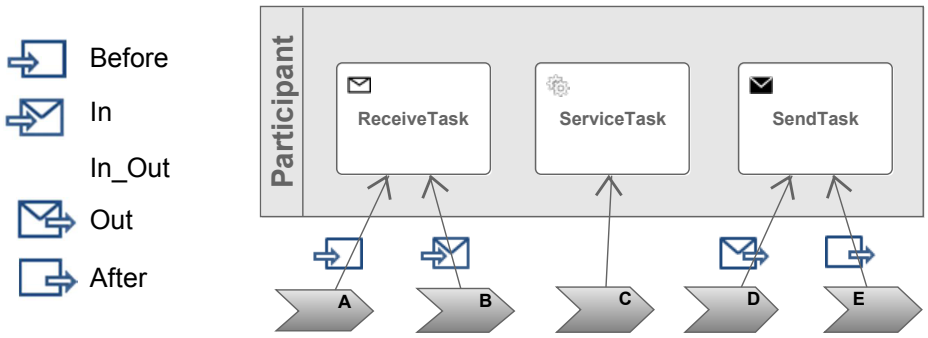

Figure 3.34: Mapping Notation for the Gray Box View

Running Example

In Figure 3.35 the mapping of actions and activities modeled in the previous two phases is shown. The semantics of this mapping is explained in the following. After receiving the message for the *receivePurchaseOrder* receive task, the *StartTimer* action and *Auth** activity are executed (no ordering restriction between them is modeled). This is achieved by a non-functional association connection of type *In*. Then, the caching actions *ReadFromCache* and *WriteToCache* are mapped to outgoing, respectively incoming, messages of the *CalculateDiscount* service task. Notice that in this case the *ReadFromCache* action must be mapped to outgoing messages instead of incoming ones, because a task is regarded from the consumer view.

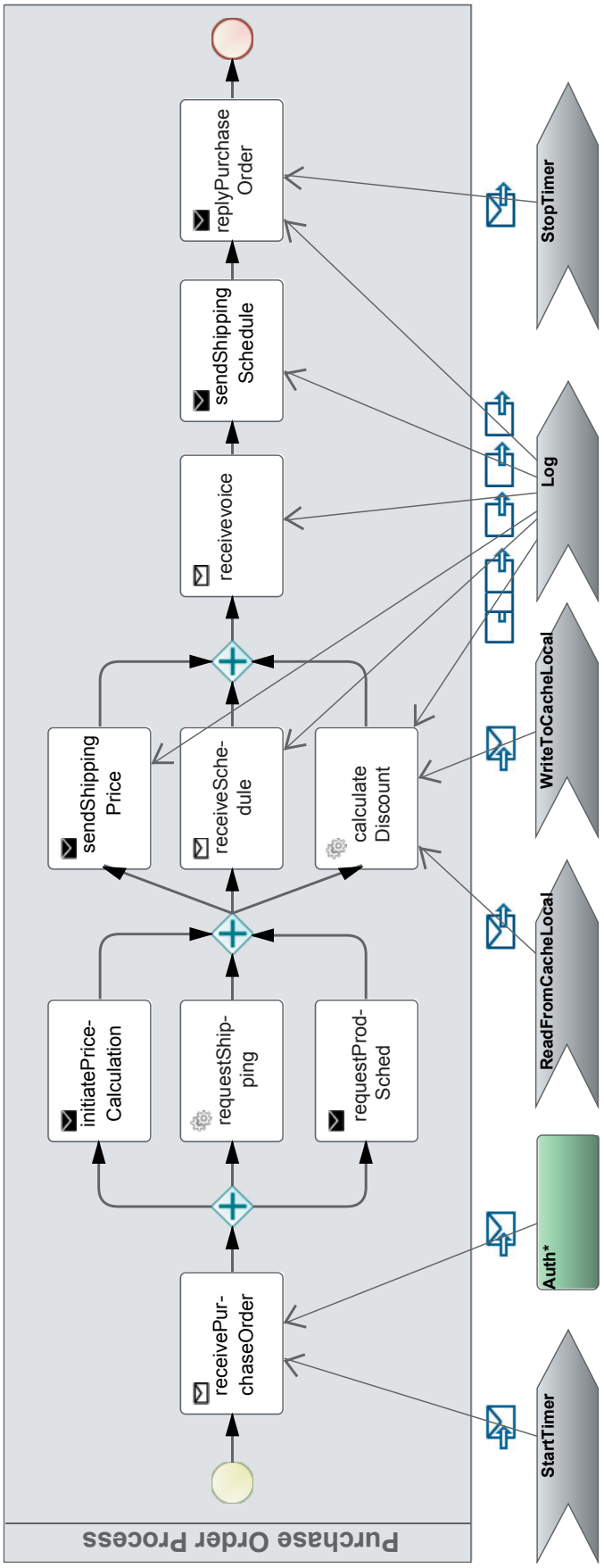

Figure 3.35: Mapping of Actions and Activities to the Purchase Order Process

The process consumes the *CalculateDiscount* service, and before it invokes it (sending an outgoing message) it should perform a cache look up. Finally, before sending the response to the consumer of the purchase order service, the *StopTimer* action is executed due to the mapping to the *replyPurchaseOrder* send task of type *Out*. Furthermore, the *Log* action is mapped after several process tasks to log their successful execution.

3.5.5 Scope-Aware Validation

This section does not describe a particular phase of the NFComp approach but rather explains the concepts of scope-aware validation of interdependencies between actions. Figure 3.36 shows both the static (design time, lower compartment) and dynamic (runtime, upper compartment) aspects of a composite web service: The BPMN process statically defines the possible control flow between tasks by using *sequence flow* edges. The non-functional association ($ASSOC$) associates these with non-functional actions (A) which, in this case, have the names *Action 1*, *Action 2*, etc. Associations are only shown for *Task 1* in the figure, although all other tasks may have associated actions, too. Both tasks and actions are executed at runtime and emit events which are themselves labeled with the task and action by which they are emitted. The association type by which actions are associated with tasks determines the order in which they are executed and, hence, the order of the respective events. Figure 3.36 shows only a sequential process; however, generally the concept is also applicable in case of control flow branching, parallelism, and loops in processes.

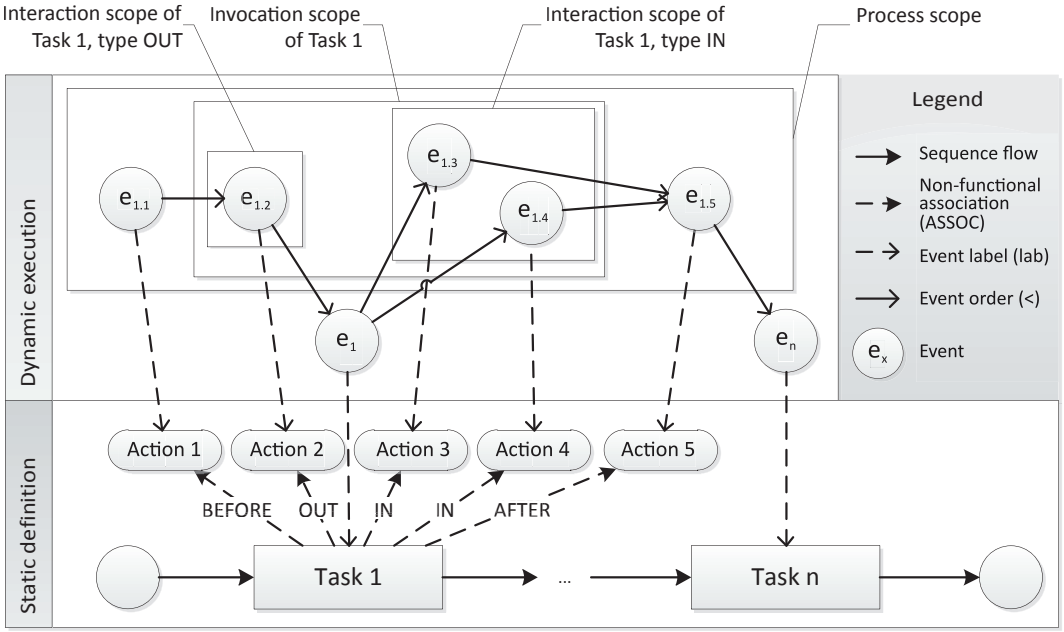

Figure 3.36: Visualization of Process, Events and Scopes

Scopes can be represented by event groups: The *interaction scope* event group for association type *OUT* of *Task 1* contains $e_{1.2}$, the only event emitted by an action which is associated

with *Task 1* by type *OUT*. For type *IN*, the event group contains the two events $e_{1.3}$ and $e_{1.4}$. The *invocation scope* comprises all the aforementioned events ($e_{1.2}$, $e_{1.3}$ and $e_{1.4}$), and the *process scope* consists of all events which have been emitted by execution actions associated with tasks of this process. It is not necessary that the event order be total: Events $e_{1.3}$ and $e_{1.4}$, for example, do not have a defined mutual order. This is, in particular, the case if Actions 3 and 4 are executed in parallel.

Although the validation mechanism provided by NFComp is static and design-time-based, it requires the concept of events. The reason is that one and the same process task or action could be executed multiple times at runtime, for instance, owing to loops in the process. Hence, such a task or action will produce events at different execution points. It is important to distinguish such events to be able to validate whether every execution of a particular action is correct with respect to the given interdependencies. For example, it could be necessary to check whether an action precedes all occurrences of another action.

3.5.5.1 Running Example

Figure 3.37 shows the mapping of the actions to the purchase order process. The goal of the modeler was to associate certain tasks with non-functional actions. The first and the last task were to be associated with *StartTimer* and *StopTimer* actions in order to measure the execution time of the process. The task *receivePurchaseOrder* was to be associated with an activity *Auth** which comprises *Authentication* and *Authorization*. The task *calculateDiscountForProduct* should be associated with *ReadFromCache* in an outgoing direction and with *WriteToCache* in an incoming direction. The purpose of these actions is to cache the discount values calculated for previous product orders for reuse in order to avoid unnecessary service calls.

There are three mistakes (shown in Figure 3.37) the modeler made in the mapping phase:

1. The modeler forgot to associate *StartTimer* with the first task.
2. The modeler associated *WriteToCache* with *receiveInvoice* and not, as intended, with *calculateDiscountForProduct*.
3. The modeler did not include *Authentication* in the *Auth** activity but only *Authorization*.

However, as the interdependency definitions do not contain any scope information, the validation procedure would assume only one particular scope for all interdependencies when the user initiates validation by hitting the *Validate* button:

Assuming process scope, the second mistake, *WriteToCache* being associated to the wrong task, is not detected because actually, *WriteToCache* is always executed after *ReadFromCache*, and thus Interdependency 2 is always fulfilled. The intended semantics of this interdependency, however, is that *ReadFromCache* requires a later *WriteToCache* action *in the same invocation scope*.

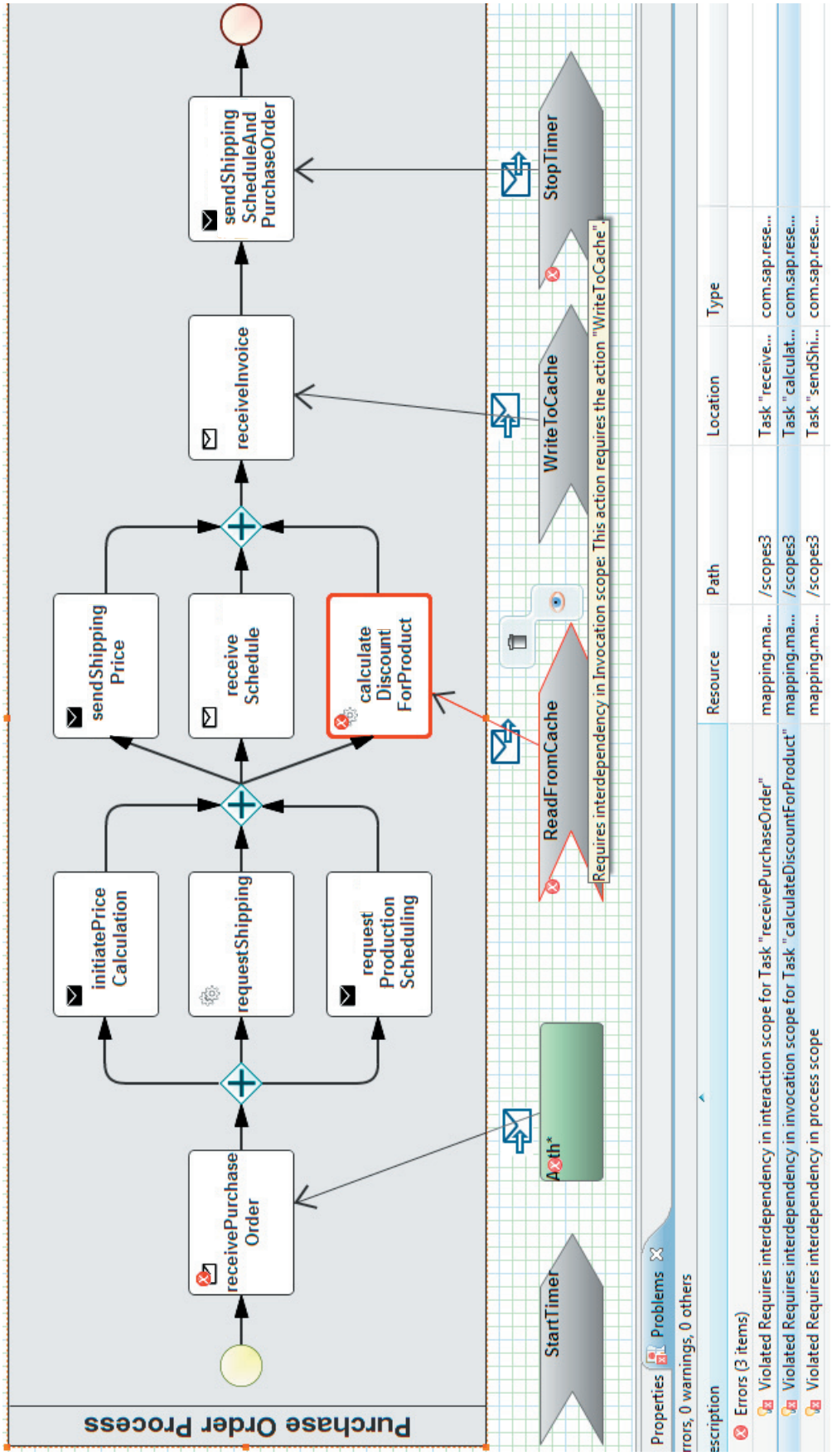

Figure 3.37: Process Validation Presented in Eclipse-based Mapping Editor

Assuming invocation or interaction scope will lead to an error although no mistake has been made: *StartTimer* is not contained in the same invocation or interaction scope as *StopTimer*; therefore, the validation procedure would misleadingly complain about this. A similar false error appears for *WriteToCache*, which is not in the same interaction scope as *ReadFromCache*.

With scopes, the interdependencies formulated in Subsection 3.5.4 can be refined as follows: 1. *StopTimer* requires an earlier *StartTimer* action *in the same process scope*. 2. *ReadFromCache* requires and must precede *WriteToCache in the same invocation scope*. 3. *Authorization* requires an earlier *Authentication* action *in the same interaction scope*.

Assuming the mistakes from above, they are correctly detected by the validation procedure as shown in Figure 3.37: An overview of all errors is presented in the *Problems View* of Eclipse. When one of them is selected, the respective elements are highlighted in the diagram. Furthermore, errors are indicated by decorators, small warning or error icons integrated in the diagram elements.

3.5.5.2 Formal Definition of the Validation Process

In order to give a precise definition of the scopes concept, a formal model for all relevant concepts is provided. As already stated in Section 3.5.2.1, three relevant scope types have been defined.

$$ScT = \{\text{INTERACTION}, \text{INVOCATION}, \text{PROCESS}\}$$

In order to specify associations from actions to tasks more precisely, association type constants AsT are used which express the point in time at which associated actions are to be executed.

$$AsT = \{\text{BEFORE}, \text{OUT}, \text{IN}, \text{AFTER}\}$$

The intuitive meaning of these values is as follows: BEFORE executes an action immediately before the task is executed in the process; AFTER is the analogous value for executing the action directly after the task. OUT is the association value to be used in order to integrate the action into a message sending task, which means that the action has access to the outgoing message; IN analogously associates the action with the receipt of a message, e.g., the response. Notice that the type IN_OUT is not in AsT because it can be represented by two associations to the same action, one with type IN and one with type OUT. Based on the association types AsT the concepts for tasks, actions and associations can be defined.

Tasks Let T be the set of functional tasks the process consists of.

Actions The set of non-functional actions is denoted by A.

Non-functional associations The set of non-functional associations is defined as $ASSOC \subseteq T \times AsT \times A$. Intuitively, $(t, at, a) \in ASSOC$ means that the functional task t is associated with the non-functional action a and the association has type at.

All elements described so far are part of a static process definition. The dynamic elements of a process, i.e., the elements which depend on a specific instance of a process are introduced in the following:

Events E denotes the set of events emitted by the process and the total function $lab : E \rightarrow T \cup A$ defines event labels. The rationale behind this is that execution of a task t is supposed to produce an event labeled with t, and analogously the execution of action a is supposed to produce an event labeled with a. Events are divided into the two disjunct sets for task events and action events, respectively:

$$\begin{aligned} E_T &= \{e \in E | lab\,(e) \in T\} \\ E_A &= \{e \in E | lab\,(e) \in A\} \end{aligned}$$

Action-event association Conceptually, an action is executed because it is associated with a functional task. The function $aea : E_A \rightarrow AsT \times E_T$ gives the mapping from action events to the task events they belong to and the association types. For this function the following must hold:

aea and the event labeling must respect the non-functional association $ASSOC$, i.e., for each association connecting a particular task with an action and each event emitted by this task, the function aea must associate the corresponding action event with the task event. Furthermore, aea may not associate more events with a task event than non-functional actions associated with the executed task:

$$\begin{aligned} \forall e_T \in E_T : \forall at \in AsT : \forall a \in A : \\ |\{t \in T | lab\,(e_T) = t \wedge (t, at, a) \in ASSOC\}| = \\ |\{e_A \in E_A | lab\,(e_A) = a \wedge aea\,(e_A) = (at, e_T)\}| \in \{0, 1\} \end{aligned}$$

Event order For a specific process instance, the order of produced events is given by the strict[8] order relation $< \subset E \times E$. For a valid process instance, an action event association function aea must exist such that the following holds:

- The non-functional associations $ASSOC$ must be respected in the sense that action events are executed at the correct point in time with respect to the association types.

[8] Strictness implies transitivity and anti-symmetry, but not totality.

Also, no functional task may be executed between a task and its associated non-functional action. Notice that the available association types depend on the type of task used in BPMN2. Only the *ServiceTask* can be associated with all available association types.

$$\forall e_T \in E_T : \forall (t, at, a) \in ASSOC : \forall e_A \in E_A :$$
$$(lab(e_T) = t \wedge aea(e_A) = (at, e_T) \wedge lab(e_A) = a) \Rightarrow$$
$$(at \in \{\text{BEFORE}, \text{OUT}\} \Rightarrow$$
$$e_A < e_T \wedge \forall e'_T \in E_T : \neg(e_A < e'_T < e_T)) \wedge$$
$$(at \in \{\text{IN}, \text{AFTER}\} \Rightarrow$$
$$e_T < e_A \wedge \forall e'_T \in E_T : \neg(e_T < e'_T < e_A))$$

- Actions associated with type BEFORE must be executed before actions associated with OUT for the same task event, and the same holds for IN and AFTER associations:

$$\forall e_T \in E_T : \forall (t, at_1, a_1), (t, at_2, a_2) \in ASSOC :$$
$$\forall e_{A1}, e_{A2} \in E_A : (lab(e_T) = t \wedge$$
$$aea(e_{A1}) = (at_1, e_T) \wedge lab(e_{A1}) = a_1 \wedge$$
$$aea(e_{A2}) = (at_2, e_T) \wedge lab(e_{A2}) = a_2) \Rightarrow$$
$$(at_1 = \text{BEFORE} \wedge at_2 = \text{OUT} \Rightarrow e_{A1} < e_{A2}) \wedge$$
$$(at_1 = \text{IN} \wedge at_2 = \text{AFTER} \Rightarrow e_{A1} < e_{A2})$$

- Action events associated with a certain task event e are required to be in the same time relation to other task events as e:

$$\forall e_{A1}, e_{A2} \in E_A : \forall at_1, at_2 \in AsT : \forall e_{T1}, e_{T2} \in E_T :$$
$$(aea(e_{A1}) = (at_1, e_{T1}) \wedge aea(e_{A2}) = (at_1, e_{T2}) \wedge$$
$$e_{T1} < e_{T2}) \Rightarrow e_{A1} < e_{A2}$$

With the previously described concepts, the precise definition of scopes can be formulated. A scope is always a certain context of program execution which is identified by the events emitted in this context. Therefore, a concrete scope is representable by a subset of events. As mentioned before, three scope types are distinguished:

Interaction A concrete interaction scope is parametrized with the task event for which the interaction takes place and the "side" of interaction, i.e., either the outgoing (OUT) or

incoming (IN) side. The interaction scope function is defined as follows:

$$s_{\text{INTERACTION}} : E_T \times \{\text{OUT}, \text{IN}\} \to \mathcal{P}(E_A)^{9} :$$
$$(e_T, at) \mapsto \{e_A \in E_A | aea(e_A) = (at, e_T)\}$$

Invocation The invocation scope comprises both interaction scopes for a task execution and is therefore only parametrized with the task event:

$$\begin{aligned} s_{\text{INVOCATION}} : E_T &\to \mathcal{P}(E_A) : \\ e_T &\mapsto s_{\text{INTERACTION}}(e_T, \text{OUT}) \cup \\ &\quad\ s_{\text{INTERACTION}}(e_T, \text{IN}) \end{aligned}$$

Process The process scope consists of all non-functional execution events of the process. It is not parametrized, but merely a constant for a given process instance:

$$s_{\text{PROCESS}} = E_B \in \mathcal{P}(E_B)$$

Further, the scope event set function is defined as follows $scopes : ScT \to \mathcal{P}(\mathcal{P}(E_B))$ which provides all concrete scopes for a certain scope type:

$$\begin{aligned} &scopes(\text{INTERACTION}) := \\ &\quad \{ s_{\text{INTERACTION}}(e_T, at) | \, e_T \in E_T \wedge at \in \{\text{OUT}, \text{IN}\}\} \\ &scopes(\text{INVOCATION}) := \\ &\quad \{ s_{\text{INVOCATION}}(e_T) | \, e_T \in E_T\} \\ &scopes(\text{PROCESS}) := \{s_{\text{PROCESS}}\} \end{aligned}$$

Static Concept	Const.	Process-specific	Dynamic Concept	Const.	Process-specific
Functional process tasks	–	T	Process events	–	E_T, lab
Sequence flows	–	–	Process event order	–	$<$
Non-functional action	–	A	Action events	–	E_A, lab
Non-functional assoc.	AsT	$ASSOC$	Action-event assoc.	AsT	aea
			Scopes	ScT	s_x, $scopes$

Table 3.11: Overview on Concepts Used in the Formal Specification

Table 3.11 provides an overview of the concepts introduced in the formal specification. Static and corresponding dynamic concepts are shown side by side. Column *Const.* contains

[9] $\mathcal{P}(X)$ gives the power set of X.

constants defined with respect to the concepts. Column *Process-specific* lists the specification elements which are specific for either the static process definition or a concrete process instance, or it contains "–" if the formal model does not consider the concept. However, the following process information is not captured by the formalization:

- the possible control flow between tasks
- the static definition of execution order of non-functional actions: This affects the order in which actions are executed that are associated with the same task and the same association type. Although not captured in the static model, the order in which actions are actually executed is captured by the dynamic part of the model, namely by the order of the emitted events.
- parallelism of events: The model does not give information about whether two events which are not ordered by the event order $<$ are emitted by tasks/actions executed in parallel or not. However, this information is not relevant for further considerations.

3.5.5.3 Leveraging Scopes for Constraint Validation

Processes are often subject to certain constraints which should be validated during design time. For example, if an action performs authorization, another action is required which performs authentication.

Constraint function. In general, a constraint c on a scope of a process instance is defined as a *constraint function* $con_c : \mathcal{P}(E) \rightarrow \mathcal{B}$ where $\mathcal{B}$ denotes the set of boolean values $true$, $false$. The semantic is as follows: The constraint c is fulfilled for the scope consisting of events E_S if and only if $con_c(E_S) = true$.

Closedness under narrowing. A constraint c is *closed under narrowing* if and only if:

$$\forall E_1, E_2 \subseteq E : con_c(E_2) = true \wedge E_1 \subseteq E_2 \Rightarrow con_c(E_1) = true$$

In other words, if a constraint is fulfilled in a certain scope, it must also be fulfilled in a smaller scope in order to be closed under narrowing.

Let $DepT$ be the interdependency types:

$$DepT = \{\text{MUTEX}, \text{PRECEDES}, \text{REQUIRES}\}$$

For each of these types, a constraint function is defined, parametrized with the targeted actions $a_1, a_2 \in A$, as follows:

$$
\begin{aligned}
con_{\text{MUTEX},a_1,a_2}(E_S) =& \neg((\exists e \in E_S : lab(e) = a_1) \wedge \\
& (\exists e \in E_S : lab(e) = a_2)) \\
con_{\text{PRECEDES},a_1,a_2}(E_S) =& \forall e_2 \in E_S : (lab(e_2) = a_2 \Rightarrow \\
& ((\forall e_1 \in E_S : lab(e_1) \neq a_1) \vee \\
& (\exists e_1 \in E_S : lab(e_1) = a_1 \wedge e_1 < e_2))) \\
con_{\text{REQUIRES},a_1,a_2}(E_S) =& (\exists e_1 \in E_S : lab(e_1) = a_1) \Rightarrow \\
& (\exists e_2 \in E_S : lab(e_2) = a_2)
\end{aligned}
$$

For simplicity, the first-order logic operators are assumed to return either $true$ or $false$, depending on whether the respective statement holds.

Note that, while MUTEX and REQUIRES are defined in a straightforward manner, the definition of PRECEDES leaves a degree of freedom which is specified as follows: If action a_1 must precede a_2 in a certain scope, then this requirement can be fulfilled by three constellations: 1. a_2 does not occur in the scope. 2. a_1 does not occur in the scope. 3. a_2 occurs in the scope and some (not each) execution of a_1 occurs before the first occurrence of a_2 in the same scope.

The given definition of interdependencies from Section 3.3.1 has been extended by a new parameter for scopes resulting in the relation $Dep \subseteq A \times DepT \times ScT \times A$. Intuitively, $(a_1, dt, st, a_2) \in Dep$ means that action a_1 has an interdependency of type dt in scope st to action a_2. This implies, for a process instance to be correct with respect to interdependencies Dep, each concrete scope must respect the interdependency. Formally, it can be written thus:

Correctness of process instance. A valid process instance in the sense of the definitions of Section 3.5.5.2 is considered *correct* with respect to a set of interdependencies Dep between non-functional actions if and only if for this instance it is

$$
\begin{aligned}
\forall (a_1, dt, st, a_2) \in Dep : \forall E_S \in scopes(st) : \\
con_{dt,a_1,a_2}(E_S) = true
\end{aligned}
$$

3.5.5.4 Scope-Specific Validation

After formalization of the scope constraints and the three interdependency types, it can be reasoned about validation. With this reasoning, validation can be applied for a specific scope. Figure 3.38 depicts an example for the previously introduced definitions visualized as sets. The set $Actions$ contains four actions which emit five events. The event $Ea1$ is emitted by Action $A1$ whereas $Ea3I$ and $Ea3II$ are emitted by Action $A3$ which has been executed twice. The different scope ScT and interdependency types $DepT$ are also depicted. The set Dep represents the concrete interdependencies $\subseteq A \times DepT \times ScT \times A$.

In the following, two types of scopes must be distinguished: the *validation scope* and the

definition scope. The *validation scope* is the scope to validate for, e.g., $Interaction1$ in Figure 3.38. The *definition scope* is the scope for which the constraint is defined, e.g., *Interaction* for the first element of Dep in the figure. If the *definition scope* matches the *validation scope*, a given constraint must be checked, i.e., if someone validates for *interaction* scope, she must consider all interdependencies defined in *interaction* scope. For the example shown in Figure 3.38, $Interaction1$ is validated for (demarcated by the dashed lines), and accordingly all interdependencies of type *Interaction* must be evaluated which contain an action (source or target) emitting one of the events in $Interaction1$. Thus, for the example, the first interdependency element must be evaluated.

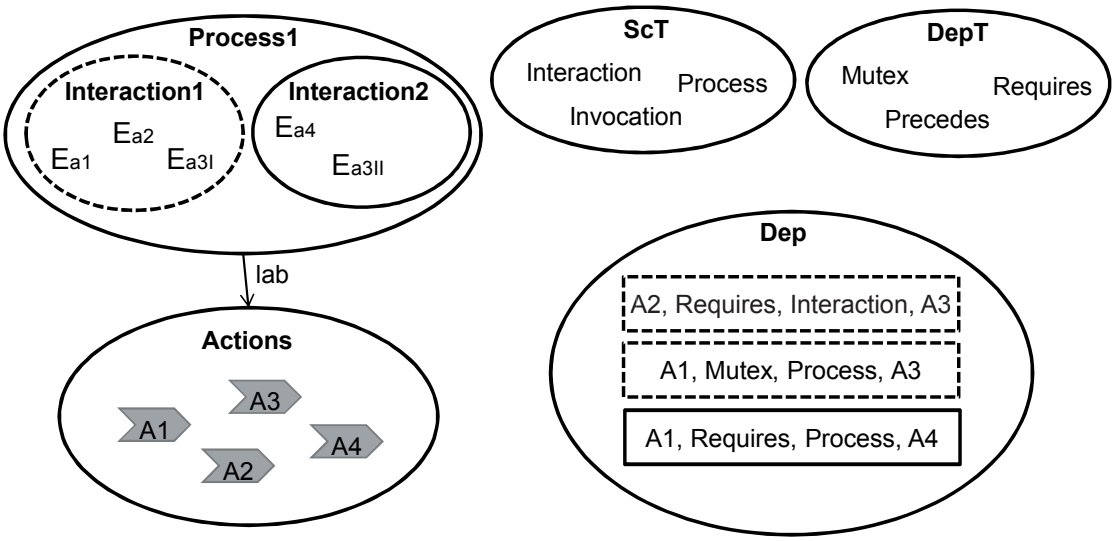

Figure 3.38: Scopes and Interdependencies Rendered as Sets

However, for particular interdependency types the rule must be extended. If the *definition scope* is a superset of the *validation scope* and the constraint is closed under narrowing, this constraint must be evaluated additionally. This is explained with the following corollary which can be deduced easily from the definition of closedness under narrowing of a constraint *c*:

$$\forall E_1, E_2 \subseteq E : con_c(E_1) = false \wedge E_1 \subseteq E_2 \Rightarrow con_c(E_2) = false$$

This means, if the validation process discovers a constraint which is closed under narrowing to be violated in a certain scope, the same constraint will also be violated in a greater scope. In the example depicted in Figure 3.38 E_1 could be set to $Interaction1$ which is a subset of the set of events of $Process1$ which would be E_2. If the constraint is closed under narrowing and there is a constraint violation in *interaction* scope, this violation will also occur in *process* scope. Consequently, in this case, the validation should additionally be performed in *process* scope taking additional interdependencies into account (the second and third element of Dep in Figure

3.38 are new possible candidates). However, only those interdependencies which impose constraints that are closed under narrowing must be evaluated. This property of interdependencies is discussed in the following.

One can show that a constraint described by the constraint function $con_{\text{MUTEX},a_1,a_2}$ is closed under narrowing, whereas this is not true for the functions $con_{\text{PRECEDES},a_1,a_2}$ and $con_{\text{REQUIRES},a_1,a_2}$. Hence, one should validate a *mutex* interdependency on a greater scope than it is defined for. This can be explained as follows: If a *mutex* interdependency between actions $A1$ and $A3$ were defined in process scope and the actions were both part of the same scope $Interaction1$, this would be a constraint violation although *mutex* was not defined in *interaction* scope. The occurrence of both actions $A1$ and $A3$ in the same interaction violates *mutex* constraints even when defined in process scope, because actions excluding each other for the whole process will also exclude each other for a single interaction or invocation. This is because a process consists of several invocations consisting of several interactions.

In contrast, this cannot be assumed for *precedes* and *requires* interdependencies. They are not closed under narrowing: If, for instance, a *requires* interdependency is violated in a particular scope, then a certain event emitted by a particular action is missing in the scope. This event, however, could be contained in a greater scope so that the interdependency would not be violated for this scope. In the example shown in Figure 3.38 the third element of Dep must not be evaluated for *validation scope* $Interaction1$, because it is of type *requires* and not *mutex*. If the element were validated, the validation would discover a violation (because $Ea4$ is missing in $Interaction1$) although the constraint is fulfilled because actually $Ea4$ is in $Process1$ scope.

3.5.5.5 Applying Scope-Specific Validation to NFComp Models

Validation is performed in the action composition and mapping phase. In the black box view non-functional activities have been validated. Because non-functional activities compose actions which are executed during one and the same interaction, the *validation scope* must be set to *interaction*. According to the rules above *requires* and *precedes* interdependencies in *interaction* scope and all *mutex* interdependencies, regardless of the scope, must be considered during the validation of non-functional activities.

In the mapping phase, actions and activities can be mapped to services (black box) or processes (gray box). For the black box mapping, the *invocation* scope must be considered, because the sets of mapped actions for incoming and outgoing messages may share particular interdependencies which must be validated. For the gray box mapping the *invocation* and the *process* scope must be validated. Figure 3.37 shows how the NFComp mapping editor presents the validation results for the different *validation scopes*. The violation of a constraint is depicted in the mapped activity and the target process task. All types of *validation scopes* are validated including the *interaction* scope which is checked for each mapped non-functional activity.

3.5.5.6 Correctness of the Validation Approach

It is desirable to prove correctness for a given process before it is executed. The procedure of performing this proof is called *static process validation*. In general, static validation is desirable for all kinds of constraints; however, NFComp is restricted to correctness with respect to constraints imposed by the defined interdependencies. A static validation procedure may fulfill one of the two following properties:

Soundness Whenever the validation procedure proves the process to be correct with respect to a set of interdependencies, then each possible run of this process is correct with respect to this set.

Completeness Whenever the validation procedure fails to prove the process to be correct with respect to a set of interdependencies, then there is a run of this process which is not correct with respect to this set.

A validation procedure which is both sound and complete is generally desirable. This, however, is in general impossible because it implies solving the halting problem, which is known to be undecidable. In order to make clear why the halting problem may be involved for a correctness proof, consider the following example: Let P be a process containing a conditional cycle (i.e., the cycle can be left at some point in time). Within the cycle, an action a_1 is executed which, according to the set of interdependencies, requires an action a_2 in process scope. This action is only executed after the cycle. Therefore, one must show that the cycle has finally been left, which is generally not possible when assuming a Turing complete process description language [90].

The implementation provided along with this thesis is at least sound. Soundness is more important for NFComp than completeness: Soundness is equivalent to the guarantee that, if a process fails at runtime, the validation procedure will not prove it correct. This guarantee is not possible if only completeness was fulfilled.

3.5.6 Action to Middleware Service Mapping

In the gray box view, actions are mapped to middleware services similar to the black box view. However, the main difference is the context information an action can access. In the black box view, only information on the service invocation is available, whereas in the gray box view there is also information on the process context. Hence, there is a set of new context variables for accessing the process context.

Table 3.12 presents and explains the new context variables. Instead of a variable, a complex expression can also be used. The language used for this expression is Xtend[10]. Xtend is a functional programming language intended for the extension of existing metatypes by additional logic. However, Xtend is also a good match to query models. A single expression can be used to

[10] `http://www.eclipse.org/xtend`

select particular attributes from model elements. For example *$currentActivity.name* will return the name of the current activity. NFComp makes use of the standard BPMN2 Ecore Metamodel provided by the Bpmn2 Project (which is a subproject of Eclipse Model Development Tools[11]) in order to query the model. The types *Activity* and *Process*, for example, represent the corresponding metamodel concepts in BPMN2. All attributes and references can be queried using Xtend. A more complex query allows obtaining for example the next activity's name by the following expression: *$currentActivity.outgoing.first().targetRef.name*.

Variable	Type	Description
$processId	String	The identifier for the intercepted process, e.g., PurchaseOrder
$processInstanceId	String	The identifier for the process instance, e.g., 1222442 which together with the processId uniquely identifies a process.
$activityId	String	The identifier which uniquely identifies an activity in a process, e.g., ServiceTask1
$currentActivity	Activity	The activity element that has been intercepted.
$currentProcess	Process	The process element that has been intercepted.
$request	XML	The request message also available for actions mapped to *out* direction. For *in* direction $message equals $request.

Table 3.12: Context Variables Available in the Gray Box View

Running Example

In the *Action to Composite Service Mapping* phase, the *Log* action was mapped to several tasks in the process. In order to log which task has been executed, additional information must be submitted to the *Log* action. The local logging mediator from Apache Synapse does not support such a scenario because it only logs SOAP messages that have been intercepted. Thus a *LoggingWebService* has been implemented offering an operation *log* with parameters *loglevel* and *logcontent*. This web service is mapped to the *Log* action. The *logcontent* parameter has been set to the expression "Activity " + $activityId + " of process " + $processId + " has been executed successfully".

3.5.7 Generation of NFC Enforcement Code

In the gray box view, a similar generation approach as in the black box view is used. It is also based on a proxy using Apache Synapse ESB. The runtime architecture is slightly different from the one used in the black box view.

As can be seen in Figure 3.39 the proxy stands in front of a WS-BPEL server executing the composite web service. One difference from a proxy-perspective between a WS-BPEL service

[11] `http://www.eclipse.org/modeling/mdt/?project=bpmn2`

and an atomic web service is that the composite web service is both at the same time — provider of a new service and consumer of other partner web services. Thus, the ESB must proxy the composite web service as well as its partner web services. For example, if the composite web service consumes three partner services, four proxies are required.

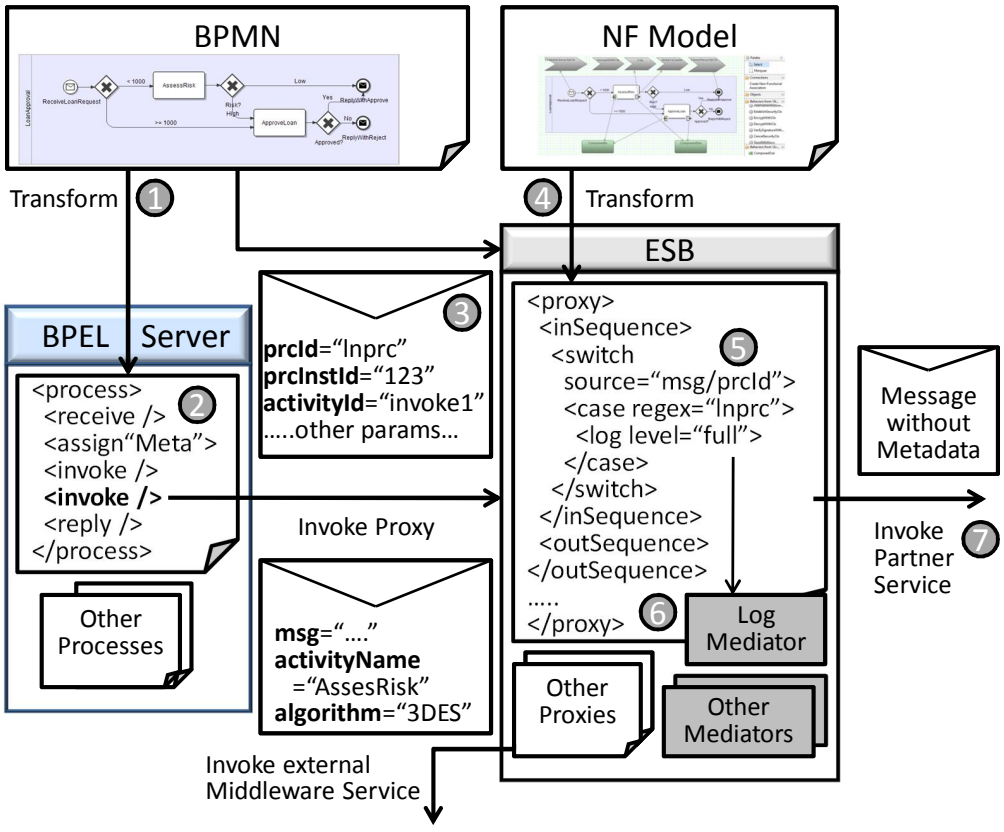

Figure 3.39: Runtime Enforcement of the Modeled NFCs

Another difference is that the proxy requires knowledge about the internal process state in order to decide if an NFA mapped in the model must be executed. This internal state must be transmitted to the proxy. For this purpose, the SOAP message which is delivered from the BPEL to the proxy is extended with the necessary metadata. The details of the runtime enforcement can be seen in Figure 3.39. The numbers in this figure depict the different steps which are explained in the following.

The generation starts with the BPMN-to-BPEL transformation (1) which is not only a classical translation from one process language into another as described in [67]. In addition, it extends the generated BPEL process in a way that the required metadata is delivered to the proxy. Moreover, the messaging activities are modified so that the sent messages pass through the proxy instead of delivering them directly to the partner web services. To accomplish this additional *assign* activities are added (2) which copy the required metadata *processId*, *processInstanceId* and *activityId* to the message which is being sent to the proxy (3). The transformation from the mapping model to the ESB configuration (4) adds a switch mediator (5) to the proxy configura-

tion checking against these metadata, for instance the mediator realizing the mapped NFA (6) is only executed if the *processId* and the *activityId* match. Before the proxy delegates the message to the target web service, it removes the metadata again (7). In case of non-messaging activities used as mapping targets, there is no such message that can be used for metadata transportation, nor is there any event that the proxy is aware of, hence additional *invoke* activities need to be added to the process. This is in order to inform the proxy about the execution and to deliver the metadata so that the proxy is able to determine the current execution context.

3.5.7.1 Overview of the Process Generation

In the following the BPMN-to-BPEL transformation depicted in (1) is elaborated in more detail. Initially, the composite web service modeled in BPMN2 must be transformed into an executable artifact since information on data flow and tasks to web service mapping are still missing. There are generally two options to accomplish this:

1. **Option A** Add data flow and service mapping information to the BPMN2 process and execute it directly on an engine capable of supporting BPMN2 processes.

2. **Option B** Add service mapping information to the BPMN2 process and transform the BPMN2 into WS-BPEL to execute it on a BPEL server. Data flow can be specified using *assign* activities in WS-BPEL.

Option A is not supported by NFComp since there was no mature BPMN2 engine available that fit to NFComp's requirements at the time when NFComp was being developed, whereas a variety of mature commercial and open-source WS-BPEL engines did exist. Instead, NFComp provides a BPEL transformation and thus supports Option B. The lacking data flow information can either be added at the BPMN2 or BPEL level because it is not required for the non-functional mapping. In order to reuse existing BPMN2-to-BPEL transformations, NFComp separates the BPMN2-to-BPEL transformation into two process transformations: BPMN2BPEL and BPEL2BPEL. BPMN2BPEL is the purely logical translation from BPMN into BPEL, whereas the BPEL2BPEL transformation extends the purely functional WS-BPEL process by non-functional processing capabilities as follows:

1. Change the service location specified in the WSDL files of the partner services to the proxy location. This will cause the BPEL process to deliver the messages to the proxy instead of directly delivering them to the partner services.

2. Modify the request message data type in the WSDL of the partner services by adding three additional XML Schema elements: *activityIdMeta*, *processIdMeta* and *processInstanceIdMeta* of type string. This is necessary to store the meta information to the request message.

3. Add an *invoke* as the first activity in the main sequence of the BPEL process in order to produce a unique and accessible instance identifier. This identifier cannot be accessed by the standard WS-BPEL language. Some BPEL engines allow access to the identifier by BPEL language extensions. Thus, NFComp uses a web service to generate this identifier. To access the identifier the generator adds an output variable and a partner link pointing to the *ProcessIdWebService*.

4. For all variables using message types being extended by the additional metadata, add the metadata elements to the *copy* element (part of the *assign* activity) responsible for variable initialization.

5. For all invoke activities add an additional *assign* activity copying the specific metadata to the corresponding variable used as input for the invoke activity. Then move the *assign* directly before the *invoke* activity. The *activityIdMeta* is a fixed value derived from the activity name, *processIdMeta* is a fixed value derived from the process name and *processInstanceIdMeta* is the value written to the output variable of the invoke for the *ProcessIdWebService* from Step 3.

3.5.7.2 Middleware Services for Composite Non-Functional Actions

In gray box view, middleware services are — as in black box view — local mediators or web services. A difference, however, is that not only atomic but also composite non-functional actions must be implemented. Middleware services realizing composite actions usually need to manage state. In NFComp, this state can either be managed inside the service itself or in the enterprise service bus. With the former strategy, an identifier must be used as additional parameter so that the middleware service is able to look up the current state (especially, during a sequence of related operations). With the latter strategy, the whole state is exposed to the ESB and must be passed with each operation. A drawback of this strategy is that the transportation overhead is relatively high, but on the other hand, other middleware services are able to operate on the exposed data, too. For example, if the transaction context is going to be exposed to the ESB, for example, the security service is able to encrypt it or even add additional data. Furthermore, referential integrity is preserved as with functional programming languages, i.e., a middleware service operation behaves as a real function where the return value only depends on its arguments. The advantage is that data dependencies can be observed just by looking at the arguments and return values of the middleware service operations. There are no side effects and thus operations can flexibly be composed similar to function composition.

3.5.7.3 Running Example

The purchase order process shown in Figure 3.30 lacks information on the mapping from tasks to web services and no data flow is modeled. Furthermore there is no information on the interface of the process. Hence, the responsible architect first designs the interface of the process in

terms of WSDL. Then, she decides to add the mapping information from services to BPMN2 tasks. She identifies a set of existing web services (*DiscountService*, *ComputePriceService*, *ShippingService* and *SchedulingService*) and imports their respective WSDL definitions into the BPMN2 editor of her choice. The editor in turn extends the BPMN2 model by adding BPMN2 *Interface* elements to the *Process* element mapping the respective WSDL port type to this interface. For each operation exposed by the *port type* of the service a BPMN2 *Operation* element is added, pointing to the respective WSDL operation. Further, the WSDL messages are transformed into BPMN2 *Message* elements. This allows mapping of the service and sending and receiving tasks to the respective service operations.

After the enrichment of the process with web-service-related information, the architect transforms the BPMN2 process into an executable BPEL process. The resulting BPEL artifact is shown in Figure 3.40, which represents a graphical view on the process using the Eclipse BPEL Designer Project[12]. The BPMN2 tasks *Receive*, *Send* and *Service* have been turned into BPEL messaging activities *receive*, *reply* and *invoke*. Sequential flow is transformed into the *Sequence* activity (*PriceCalc*, *Shipping*, *Scheduling*, *Discount* and *Scheduling*) and parallelism through gateways is expressed through *Flow* activities (the boxes with the bold borders on top and bottom). To define the data flow between messaging activities, the *assign* activity is used in BPEL. It allows the initialization of variables (demarcated by the small I in the graphical representation) and copying of data from variable to variable. A messaging activity then consumes this variable, either by reading from it or writing to it.

Additionally, the architect defines the necessary *port types*. Then, the *partner links* need to be specified in order to define the communication from the process to its partners, respectively the clients, consuming the process. Finally she creates a deployment descriptor (in case of Apache ODE the *deploy.xml* file) for the association of partner links in the process with concrete services and ports.

In order to deploy the process, the BPEL definition, the deployment descriptor and the WSDL files of the partners as well as the process itself are bundled into an archive, which is loaded by the BPEL server and then compiled into executable Java code. The architect tests the process against the business requirements.

To expose the necessary information on the execution context needed by NFAs, the BPEL process must be instrumented. Hence, the NFComp code generator responsible for the BPEL2BPEL generation does the 5 transformation steps described above. The results can be seen in Figure 3.41.

[12] `http://www.eclipse.org/bpel/`

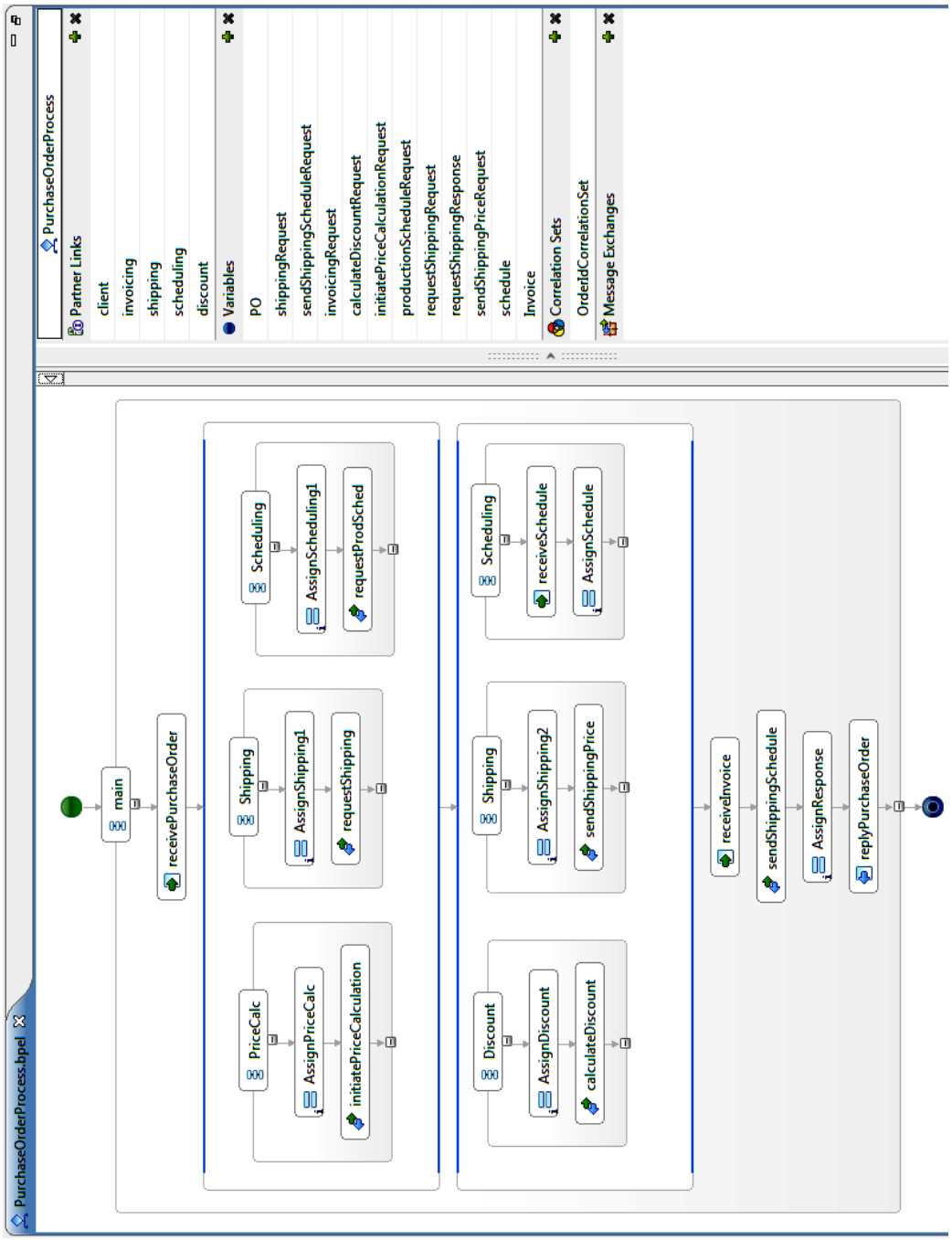

Figure 3.40: The Purchase Order BPEL Process Shown in the Eclipse BPEL Editor

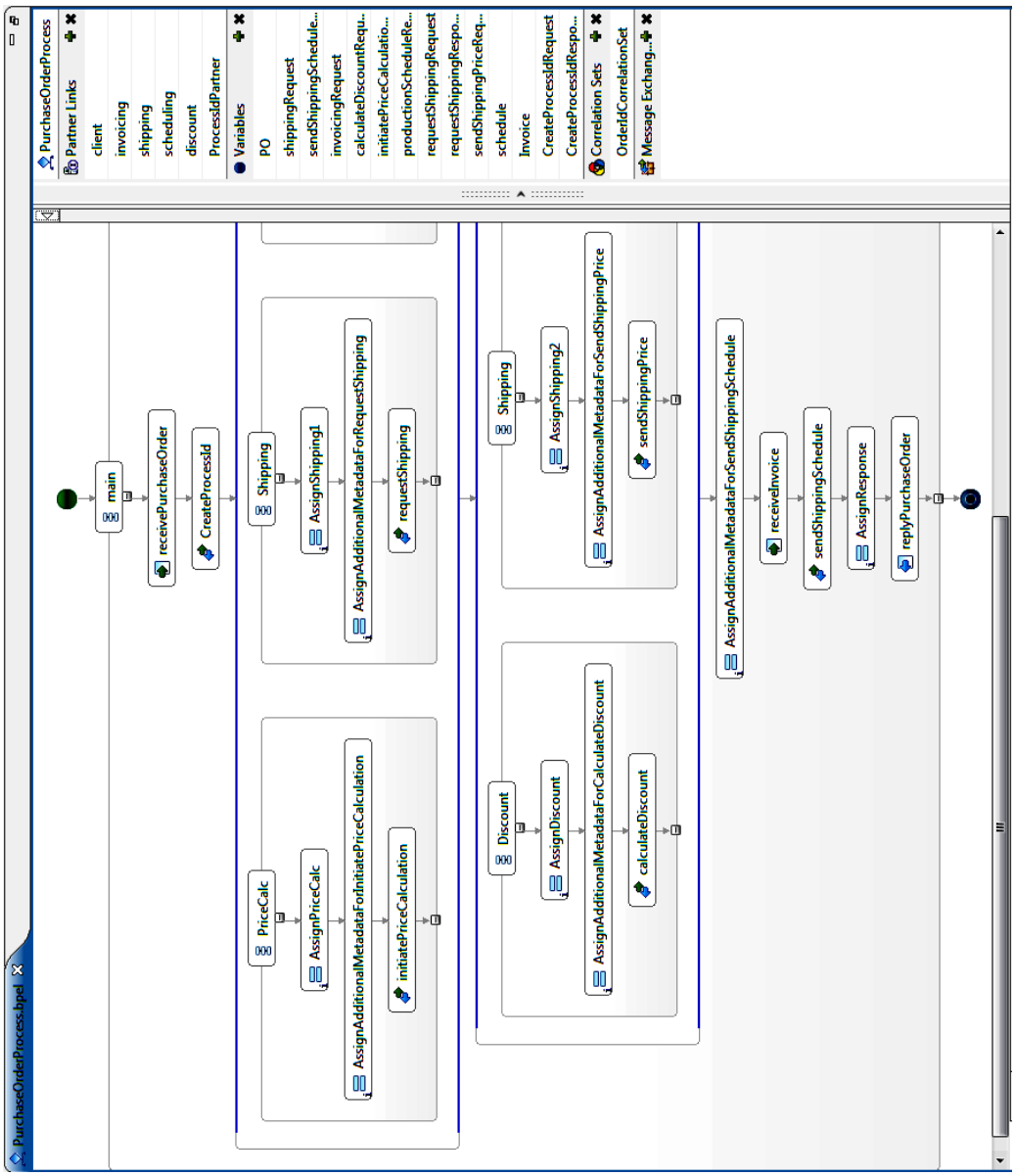

Figure 3.41: The Purchase Order BPEL Process After the Transformation

A new process partner called *ProcessIdPartner* and two variables *CreateProcessIdRequest* and *CreateProcessIdResponse* for storing the input, respectively the output, message have been introduced. The call to the *ProcessIdService* is performed by the *invoke* activity named *CreateProcessId*, which is executed directly after the first *receive* activity. For each assign activity which is initializing variables of a message type (marked by the small "i") additional metadata initialization is done. This cannot be seen in the graphical representation. Thus, Listing 3.5 shows the corresponding XML snippet for the *assign* activity of *InitiatePriceCalculation*.

```
1  <assign name="AssignPriceCalc" validate="no">
2    <copy>
3      <from>
4        < literal >
5          <tns: initiatePriceCalculation >
6            <tns:customerInfo>
7              ...
8            <tns:purchaseOrder>
9              ...
10           <ns1:activityIdMeta/>
11           <ns1:processIdMeta/>
12           <ns1:processInstanceIdMeta/>
13         </tns: initiatePriceCalculation >
14       </ literal >
15     </from>
16     <to part="parameters" variable=" initiatePriceCalculationRequest "/>
17   </copy>
18   <copy>
19    ...
20 </assign>
```

Listing 3.5: XML Snippet for Initializing the Request Variable of InitiatePriceCalculation

Furthermore, there is a new *assign* activity for each *invoke* activity copying the additional metadata to the outgoing message. An example is shown in Listing 3.6. There are three *copy* elements for the three types of metadata. The first one (Line 2-7) copies the name of the *invoke* activity to the *activityIdMeta* parameter. The second (Line 8-12) copies the process name to the *processIdMeta* parameter, and the third *copy* (Line 13-20) copies the returned *process id* stored in the *CreateProcessIdResponse* variable to the *processInstanceIdMeta* parameter.

```
1  <assign name=" AssignAdditionalMetadataForInitiatePriceCalculation ">
2    <copy>
3      <from>< literal >initiatePriceCalculation</ literal ></from>
4      <to part="parameters" variable="initiatePriceCalculationRequest">
5        <query queryLanguage="...: xpath1.0">tns:activityIdMeta</query>
6      </to>
7    </copy>
8    <copy>
9      <from>< literal >PurchaseOrderProcess</ literal ></from>
10     <to part="parameters" variable="initiatePriceCalculationRequest">
11       <query queryLanguage="...: xpath1.0">tns:processIdMeta</query></to>
12   </copy>
13   <copy>
14     <from part="parameters" variable="CreateProcessIdResponse">
```

```
15        <query queryLanguage=" ...: xpath1.0"><![CDATA[processidns:return]]></query>
16      </from>
17      <to part="parameters" variable="initiatePriceCalculationRequest">
18        <query queryLanguage=" ...: xpath1.0">tns:processInstanceIdMeta</query>
19      </to>
20    </copy>
21  </assign>
```

Listing 3.6: XML Snippet for *AssignAdditionalMetadataForInitiatePriceCalculation*

The resulting modified BPEL process is deployed on the BPEL server and will execute the business process delivering the necessary process context information to the proxy, which in turn invokes the corresponding middleware services to enforce the NFComp model.

In the next step, the architect uses the NFComp code generator to generate the ESB configuration according to the non-functional model. Similar to the generation process in the black box view, a set of sequence configurations for the action realization is generated. The main difference to the black box approach is that for each web service consumed by a process task, a separate proxy configuration is generated. The proxy is only generated, if the task is connected to an NFA via an association. Listing 3.7 shows the resulting proxy configuration for the *DiscountService*. Note that in case of multiple process tasks consuming the same web service, a single proxy configuration is generated including multiple *switch* mediators.

```
1  <proxy xmlns="http :// ws.apache.org/ns/synapse" name="DiscountService">
2    <target>
3      <inSequence>
4        <collectctx/>
5        <switch source="get-property('processIdMeta')">
6          <case regex="PurchaseOrder">
7            <switch source="get-property('activityIdMeta')">
8              <case regex="calculateDiscount">
9                <sequence key="calculateDiscount_ReadFromCache"/>
10             </case>
11           </switch>
12         </case>
13       </switch>
14       <send>
15         <endpoint><address uri="http ://.../ DiscountService/"/></endpoint>
16       </send>
17     </inSequence>
18     <outSequence>
19       <switch source="get-property('processIdMeta')">
20         ...
21       <send/>
22     </outSequence>
23   </target>
24   ...
25 </proxy>
```

Listing 3.7: Proxy Configuration for *DiscountService*

In this proxy configuration the *collectctx* mediator (Line 4), provided by NFComp, is used. This mediator is responsible for extracting the metadata sent by the instrumented BPEL process. The extracted metadata is stored in Synapse variables which are available for the duration of the whole service execution; e.g., the *processIdMeta* data from the *DiscountService* request message is stored in a variable with the name *processIdMeta*. After storing the metadata, the context collector mediator removes it from the request message. This is necessary because the *DiscountService* should not be aware of this data and may even reject it otherwise.

In the produced files containing the mediator sequences for activities, the *caching* XML element is used to execute NFComp's caching mediator. The caching mediator is an example of a middleware service which manages the state on its own. To refer to this state an appropriate key is required, which is in this case the body of the request message. Passing this body to the *readFromCache* operation allows the retrieval of the stored response.

3.6 NFC Composition in a White Box View of Web Services

In this section, NFComp is presented from white box view. Regarding web services from white box view generally implies that all implementation artifacts for the service are visible. Thus, theoretically, the source code of the service can be accessed directly. However, web services could have been implemented in different programming languages, making a solution — directly based on the source code — platform-dependent. As a consequence, NFComp uses an abstraction of the source code by generating a platform-independent behavioral model from the code. This behavioral model is used to define a mapping from non-functional actions to the web service. In contrast to the black box view, this allows the application of actions to internal parts of the service not visible in the WSDL interface. Compared to the gray box view, the white box view allows the application of NFAs to internals of atomic web services. This enables the definition of fine-grained mappings from NFAs to web services.

This approach is not only applicable to web services but to Java-based software in general. The only restriction is that the software must be component-based in the sense that there must be a well-defined interface. This interface should expose one or more operations which are the entry points for the component triggering its execution. In the case of web services, this would be the operations defined in the WSDL interface.

Regarding the NFComp process model, the white box view introduces a new phase, *Behavioral Model Generation*. Furthermore, the *Action to Service Mapping*, *Action to Middleware Service Mapping* and *Generation of NFC Enforcement Code* phases must be extended. Figure 3.42 depicts the extended process model from Figure 3.14 with the new phase represented by the white box in the top right corner.

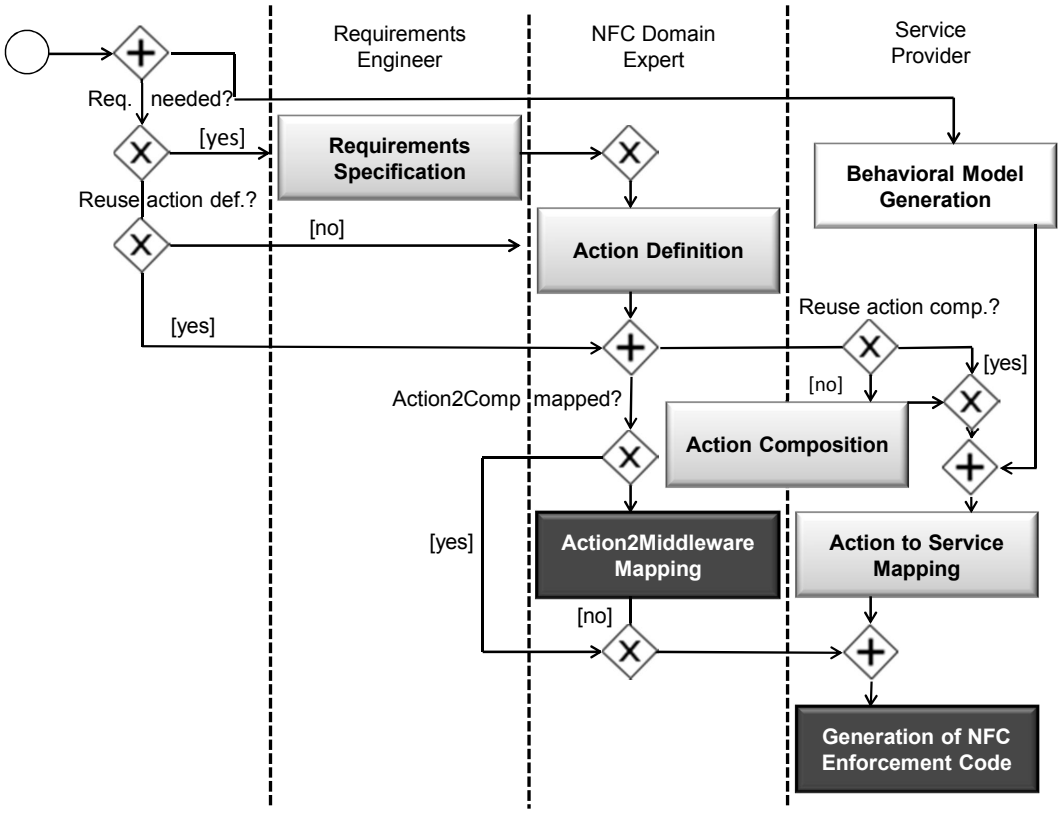

Figure 3.42: NFComp Process for Web Services from the White Box View

3.6.1 Generation of Behavioral Model from Code

The NFComp approach depends heavily on graphical models and consequently requires a visual model for the mapping of non-functional actions. In black box view, this model is an abstraction of the WSDL interface, whereas in gray box view, the composite service description in BPMN2 is used. In white box view it is assumed that there is a component which has been implemented in an appropriate programming language and its code is fully accessible. The component could either be composite or atomic, but there is usually no model that can be used for the mapping.

In order to close this gap, a generator which takes the code as input and generates a BPMN2 behavioral model as output is needed. NFComp provides a generator for Java code. However, NFComp can easily be extended by further generators for other programming languages. The resulting BPMN2 model can then be used in the mapping phase to attach non-functional actions before, after, or even around (Java) methods. In this transformation the complete, underlying Java code is processed and turned into the target behavioral model. However, those details which are relevant for concern composition can be selected. Hence, the graphical representation of the complete BPMN process abstracts from unnecessary details of the implementation. Still, the responsibility for the selection of the relevant parts of the code is left to the application provider. This is because she is the one who wants to integrate non-functional concerns into her components and actually knows which parts are relevant.

BPMN2	**Java**
Participant/Pool	Method body showing method invocations
Task	Method invocation
OR Gateway	If statement
Conditional Sequence Flow	If condition
Start/End Event	Start/End of a method

Table 3.13: Mapping Java to BPMN2

A coarse-grained mapping overview of Java to BPMN2 concepts is given in Table 3.13. The algorithm for generating the BPMN2 model is given in Listing 3.8.

```
1  for each class cl do
2    if cl implements component
3      then for each method m do
4        if method part of interface
5          then let participant = new BPMN.Participant
6            participant.name = <cl.name>.<m.name>()
7            let start = BPMN.StartEvent, let node = start
8            for each statement st in method do
9              let resultNode = call processStatement(node, st, null)
10             if(resultNode != null)
11               node = resultNode
12           let end = BPMN.EndEvent
13           connect node with end via SequenceFlow
14
15 define processStatement(Node start, Statement st, Condition cond):
16   if st is methodinvocation(object o, method inv) and !alreadyprocessed and codeaccessible
17     then let subproc = new BPMN.Subprocess
18       subproc.name = <o.class.name>.<inv.name>()
19       let methodef = find methoddefinition using methodinvocation
20       let substart = BPMN.StartEvent
21       let node = substart
22       for each statement stmnt in methodef do
23         let resultNode = call processMethodContents(node, stmnt, null)
24         if(resultNode != null) then node = resultNode
25       let end = BPMN.EndEvent
26       connect node with end via SequenceFlow
27       connect start with subproc via SequenceFlow(cond)
28       return subproc
29   if st is ifstatement
30     then let ex = new BPMN.ExclusiveGW
31     connect start with ex via SequenceFlow
32     let exMerge = new BPMN.ExclusiveGW
33     for each ifblock(condition cnd) do
34       let node = ex
35       for each statement stmt in ifblock do
36         let result = processStatement(node, stmt, cnd)
37         if(resultNode != null) then node = resultNode
38       connect node with exMerge via SequenceFlow
39     return exMerge
```

Listing 3.8: Algorithm for the Mapping from Java to BPMN2 in Pseudo Code

Firstly, for each method, starting from the top level operations (operations visible in the interface), a BPMN participant is generated with name <classname>. <methodname>(). For each method invocation in its method body, a subprocess task is generated with name <classname>.<methodname>() of the invoked method. If an *if* statement is found, an exclusive gateway will be generated, and for each *if/else* block, a new branch is generated. In these branches the method invocations are again turned into tasks. A *Start* and an *End* event are generated, connecting to the first, respectively the last, *task*. All *tasks* are connected via *sequence flows*. For *if* and *else if*, condition expressions are generated for the corresponding *sequence flow*, whereas *else* is turned into a *default sequence flow*. This algorithm is repeated recursively for each subprocess, using the method representing the subprocess as input. If the method already has a representation as a BPMN2 participant, it is not further processed (avoiding infinite loops during processing).

In order to avoid unnecessary complex diagrams, the subprocesses used in the pools are not expanded, but their contents are shown in another pool (a pool visualizes a participant in the model). Furthermore, only those pools are rendered which represent methods previously selected by the user.

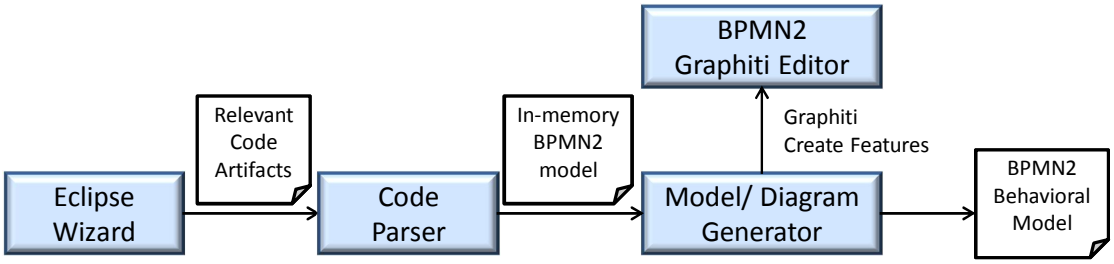

Figure 3.43: The Generation Process of Java Code to BPMN2

Figure 3.43 shows the process of generating the BPMN2 behavioral model from Java code. Firstly, an Eclipse wizard is used to select the classes and their methods that are relevant for NFC integration. Then, a code parser takes the given code input and transforms it — according to the algorithm described above — into a BPMN2 in-memory model. This in-memory BPMN2 model is just a data model without a graphical representation. This data model is processed to create the graphical representation of the process. For this purpose a Graphiti-based BPMN2 editor is used. In Graphiti editors, there are different *CreateFeature* classes which are responsible for creating a new model element and the graphical representation of this model element. The *CreateFeature* classes are usually instantiated by dragging elements from the palette into the diagram canvas. In the generation process, the in-memory model is parsed and the respective *CreateFeature* classes are called programmatically. The BPMN2 Graphiti editor in turn creates the diagram and the corresponding model elements which are stored in separate files. The graphical elements are stored in a diagram file, whereas the model elements are stored in a model file. This separation is important for NFComp, because the model file will contain the complete representation of the source code, whereas the diagram contains the graphical representation of a subset of these

model elements. More specifically, the diagram contains only those element which have been selected via the Eclipse wizard by the user.

Running Example

Listing 3.9 shows a snippet of the Java implementation of the *FlightReservationService* which has already been introduced as running example for the black box view. This example is reused and an internal view on the web service is taken. Web services are, however, only one possible component-based technique which can been chosen.

In the listing, two methods, *bookFlight* (Line 2-9) and *isReservationPossible* (Line 10-14), are shown. The *bookFlight* method is exposed as a web service operation and is thus also visible in the WSDL of the service. The *isReservationPossible* method is a private, non-visible method. It cannot be used from the black box view, because it is not part of the WSDL. The service provider decides to add caching actions to her web service. However, it is not useful to do this on the *bookFlight* method level using a SOAP message-based caching method. One reason for this is that the *bookFlight* method is not only responsible for the checking of credit card validity but also finally does the booking. Hence, *bookFlight* changes the state of the web service and cannot be cached. It is much more efficient to do the caching on a more fine-grained level. It makes more sense to cache only the calls to the credit card provider, i.e., to store the results of the validity check by the method *isCreditCardValid* in a local cache. This allows the retrieval of the results of this check for the same customer again and again without calling the credit card provider unnecessarily.

```
1  public class MyFlightReservationService {
2    public String bookFlight(String id, String customerName, String creditCardInformation){
3        boolean isReservationPossible = isReservationPossible (id, customerName, creditCardInformation)
4        if( isReservationPossible ){
5          doBooking();
6          return "booking flight #" + id + "# successful";
7        }else
8          return "Not possible to book flight #" + id + "#";
9    }
10   private boolean isReservationPossible (String id, String customerName, String creditCardInformation){
11     return FlightDatabaseAccess. isFlightAvailable (id)
12       && CustomerRelationShipAccess.isCustomerValid(customerName)
13       && CreditCardProviderAccess. isCreditCardValid ( creditCardInformation );
14   }
15 }
```

Listing 3.9: Looking into the *FlightreservationWebService* Code

In Figure 3.44, the Eclipse Wizard for the selection of the relevant methods and classes is shown. The user selected the *FlightReservationService* class, the three methods *searchBestFlight*, *searchFlights* and *bookFlight* which have been exposed as web service operations and the private method *isReservationPossible* in order to optimize the caching for the web service.

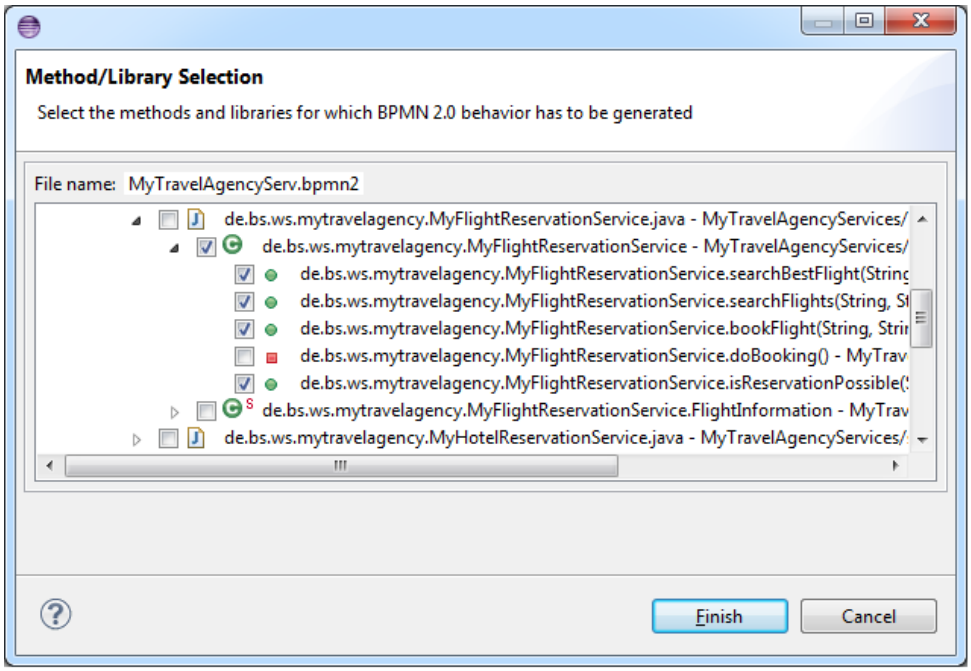

Figure 3.44: Selection of the Relevant Methods and Classes

The user could have selected other methods alternatively. There is no automatism which preselects methods which are visible in an WSDL interface (to avoid dependencies on web services).

Figure 3.45 depicts the resulting generated behavioral model (*searchBestFlight* and *searchFlights* methods omitted). Each pool represents a particular selected method containing a set of tasks representing the method invocations from within this method. The first pool represents the *bookFlight* method checking if the reservation is possible by calling the *isReservationPossible* method. The *if* statement is represented by the *exclusive gateway* using the *isReservationPossible* variable to decide whether the following tasks will be executed. Finally, the *doBooking* method is called. The *isReservationPossible* method is represented by the second *pool*. It defines a sequence of three method calls to *isFlightAvailable*, *isCustomerValid* and *isCreditCardValid*.

3.6.2 Action to Service Mapping

In the *Action to Service Mapping Phase* the generated behavioral BPMN2 model and diagram can be used for the mapping of NFAs. The subjects NFAs are mapped to are similar to those in the gray box view. However, in the white box view, tasks do not represent communication with partner web services but rather Java method invocations. Moreover, a pool does not represent a composite web service but rather a method, and the flow elements in the pool represent the method body, more specifically the sequence of method invocations. This means, that if an NFA is mapped to a BPMN2 task and thus a particular method invocation, this mapping is only valid for the invocation of the method in the context of the method represented by the pool.

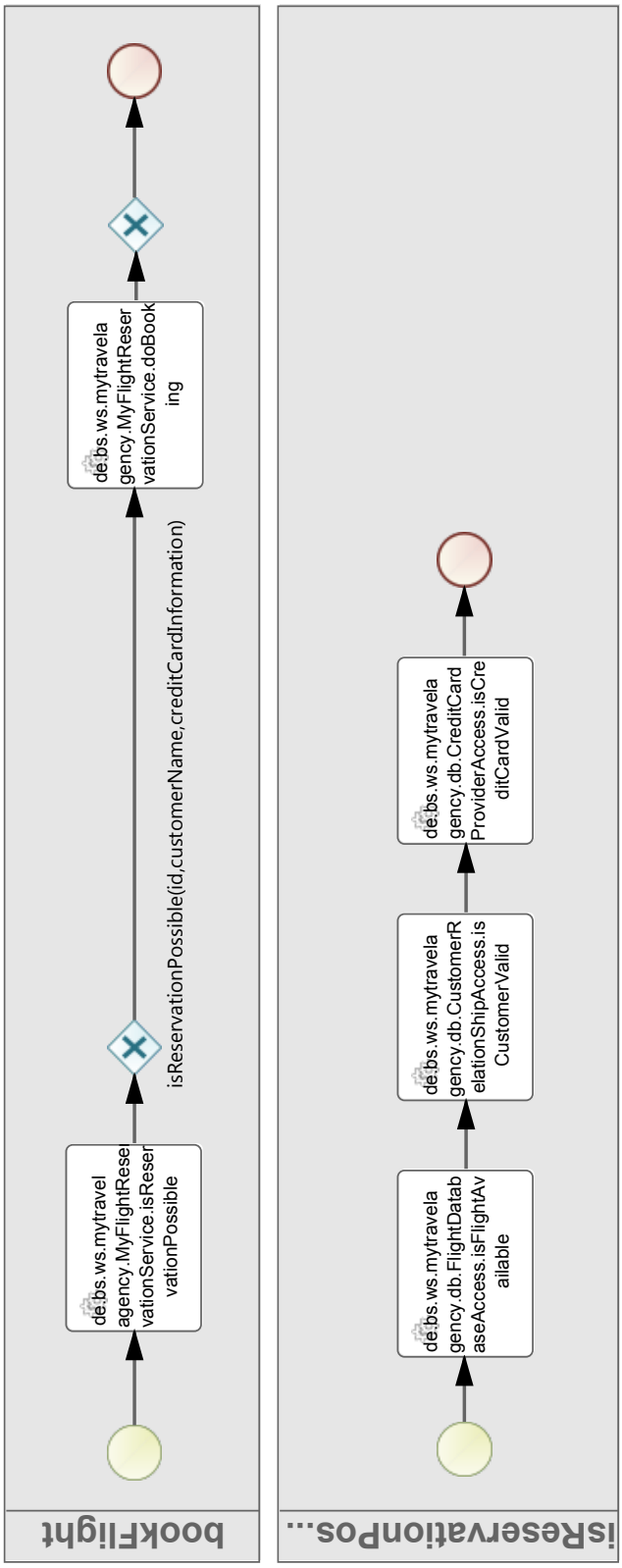

Figure 3.45: Generated Behavioral BPMN2 Diagram

Subject	Description
Before Method	An action is executed before the method is executed.
After Method	An action is executed after the method is executed.
Around Method	An action is executed and in this action the surrounded method is invoked.

Table 3.14: Subjects of NFAs in the White Box View

Table 3.14 shows the possible subjects NFAs can be mapped to. The graphical notation (cf. Figure 3.34) of these subjects is a non-functional association with the corresponding type. Before/after is represented by the before/after symbol already used in gray box mapping (the line with an arrow pointing into, respectively to the outside, of a box). The *around* type is supported by the *In_Out* association type (the line without a symbol).

Running Example

As already mentioned, it makes more sense to cache the internal *isCreditCardValid* method instead of the publicly exposed web service method *bookFlight*, because the *bookFlight* method actually does the credit card check and the booking. The booking, however, modifies data by creating a booking entry in the database and must not be cached. The behavioral BPMN2 model can be imported (analog to the gray box approach) into the mapping editor, and the caching activity can be mapped to the desired method. Figure 3.47 shows this scenario. The modeler has mapped the *CachingActivity* around the *isCreditCardValid* method and the *Log* action after this method. Figure 3.46 shows the *CachingActivity*. This activity uses the *Proceed* Action which is available for all non-functional activities. This action represents the execution of the intercepted *join point* method. In this case, the *ReadFromCache* action is executed and the returned result is stored in the *returnValue* data item. The exclusive gateway does the following: The *Proceed* Action is executed only when the *returnValue* is not equal to *null* and the return value is stored in a variable. After the *Proceed* Action, the *WriteToCache* action is executed using the *returnValue* as input to store it in the cache. Finally, the *returnValue* is additionally exposed as BPMN2 *DataOutput* (demarcated by the black arrow in the data item symbol), which means that the data is used as a return value for the intercepted join point method.

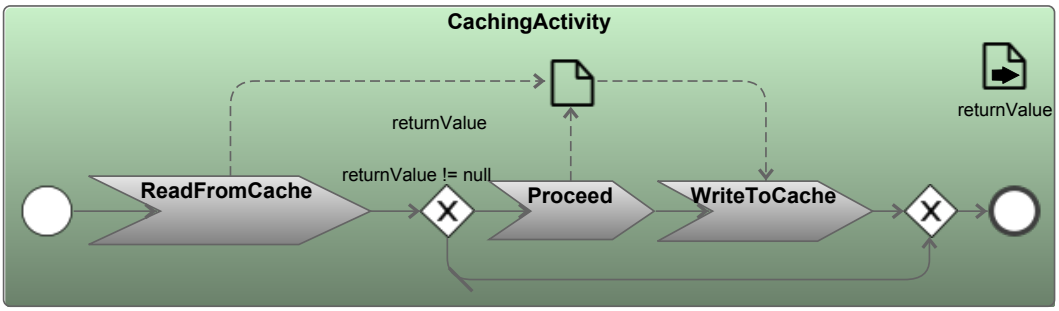

Figure 3.46: Activity Definition for Caching

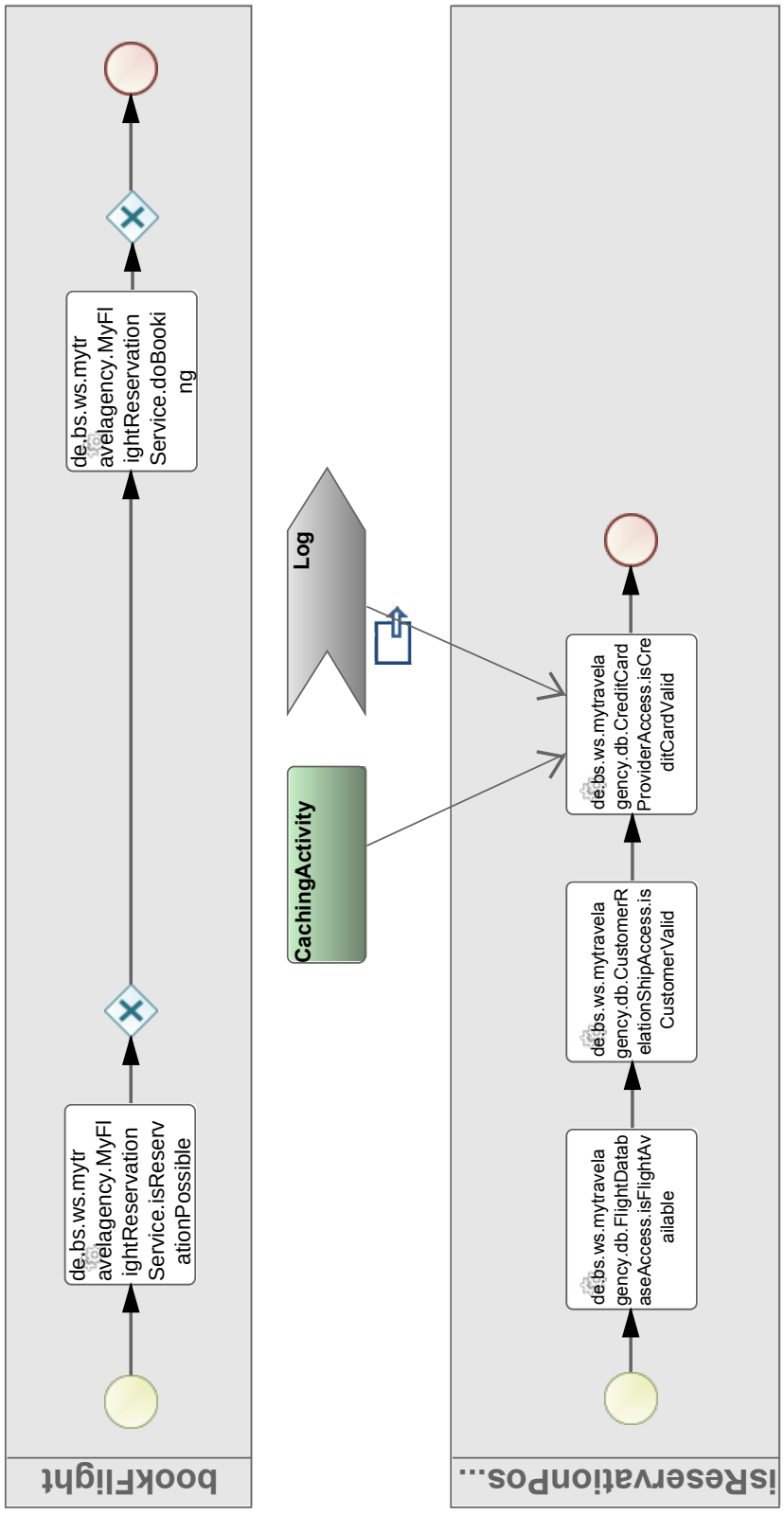

Figure 3.47: Mapping of Caching Actions in the White Box View

3.6.3 Action to Middleware Service Mapping

In the action to *Middleware Service* phase, the mapping of actions to Java-based middleware services is performed. It is assumed that there is a Java *Class* for each middleware functionality. A non-functional action maps to a single method exposed by the Java *Class*. The available mapping options that can be defined in the model are summarized in Table 3.15.

Attribute	Description
localName	The class name of the middleware service.
operation	The name of the operation implementing the middleware functionality.
method	The targeted event, either method call or method execution.
type	The type of the middleware invocation. Normal (default) or reflective. Reflective means that the join point method is not executed by the advice itself but the middleware service is invoking the method using Java reflection.
onFailure	*Resume* (default) or *terminate* are allowed values. Specifies what happens if middleware service execution results in failure. When resume is chosen, the error will not influence the web service execution. In case of terminate, the error is delegated to the web service execution.

Table 3.15: Mapping Options in the White Box View

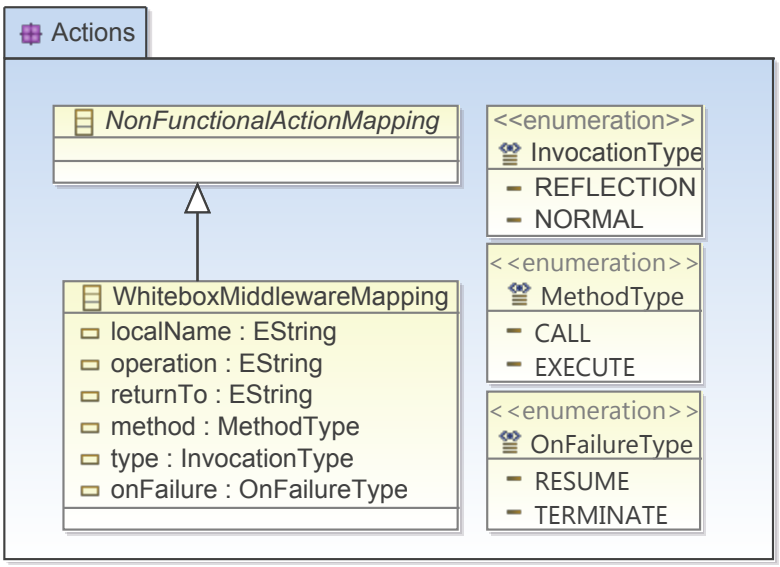

Figure 3.48: The Extended Middleware Mapping Metamodel for the White Box View

Figure 3.48 shows the extended middleware mapping in the mapping metamodel. A new subclass of *NonFunctionalActionMapping* called *WhiteboxMiddlewareMapping* has been introduced, extending its super class by the attributes defined in Table 3.15. It also inherits the characteristic that it may define a *NonFunctionalActionConfiguration* with a list of *ConfigurationEntry*

elements, which can be used to add parameters to the mapped middleware service operation. In this process, similar to the black or gray box view, predefined context variables can be used which are listed in Table 3.16. However, one difference to the black and gray box view is that the keys of the parameters should be a concatenation of the string *param* and the position of the parameter in the method signature, for example *param0* for the first parameter. This is necessary because in Java, not the name but the position of the parameter is used for method invocation.

Variable	Type	Description
$thismethod	Method	The method object intercepted by the action. This object can for example be used by the action to reflectively invoke it.
$thismethod.args	Object[]	The input parameters of the intercepted method.
$thismethod.target	Object	The object on which the intercepted method has been invoked.

Table 3.16: Context Variables Available in the White Box View

Running Example

The *ReadFromCache* and *WriteToCache* actions are mapped to one and the same middleware service. Hence, the *localName* attribute of the *WhiteboxMiddlewareMapping* is set to $de.bs.CachingService$, which is the full qualified class name of the middleware service implementation. The operation parameter is set to *readFromCache*, respectively *writeToCache*. For the *WriteToCache* action there is a key-value pair *param1* with $value = \$thismethod.args$ and *param2* with $value = \$thismethod$ passing the method object to the middleware service. The third parameter is defined by the corresponding *CachingActivity*, which defines data items mapped to the action. The data item *returnValue* has been set as the incoming data via a data association. Notice that additional parameters specified by data items are limited to one data item and this data item is always mapped to the last parameter. *ReadFromCache* defines the same parameters except the third. The *Log* action is mapped to the *LoggingService*, more specifically to its operation *log*. The key-value pair $param1 = \$thismethod$ is used to pass the invoked method to the logging method in order to track which method has actually been invoked.

3.6.4 Generation of NFC Enforcement Code

In this phase, AspectJ code is generated in order to support separation of concerns not only at the modeling but also at the code level. This means advice are generated in order to encapsulate the code for the invocation of the middleware services. Pointcuts are used to weave those invocations into the execution of the service. In the following, the mapping of model elements to AOP constructs is explained:

- **BPMN Task:** A BPMN Task represents a method invocation. The name of the BPMN task is equal to the fully qualified class name plus the method name. This is used as the

basis for generating the pointcut expression which is * <*taskname*>*(..)*. This means, independently of the return value or parameters of the method, the method named <taskname> is always selected.

- **BPMN Pool:** The BPMN pool represents a particular method declaration and thus has the same (fully qualified) name as this method. The pool contains a set of tasks representing method invocations in the context of this method declaration. Hence, each task has the name of the invoked method. In this regard it is important to ensure that the pointcut matches only for the invocation according to its context. The context pertains to two things: the execution hierarchy and the exact sequence of invocations. If, for example, a method *a()* is declared which contains, among other items, an invocation of method *b()*, then only this very invocation should be matched by a pointcut. More specifically, a pointcut should not match an invocation of *b()* from another method *c()*, but only from *a()*. To achieve the desired precision of pointcuts for the execution hierarchy, AspectJ offers the *withincode pointcut*. However, the *withincode* pointcut can only be used in combination with *call* semantics. If *execution* semantics is used instead or if there is a whole hierarchy of pools, AspectJ offers *cflow(...)* or *cflowbelow(...)* pointcuts which restrict the execution of an advice to the control flow of a particular join point.

- **Non-functional action:** A non-functional action is implemented by an operation of a middleware service. A middleware service in the white box view is represented by a particular Java class and a specific public method of this class. To call this method before, after or around the target method invocation, the non-functional action is mapped to an advice. The advice type is determined by the respective non-functional association type. The code for the invocation of the middleware service is generated from the *localName* and *operation* attribute defined in the Action to Middleware Service Mapping Phase: <*localName*>.<*operation*>(<*keyvalpair1.value*>, ... , <*keyvalpairN.value*>); For handling errors caused by the middleware service, a surrounding try-catch block is generated, and depending on the *onFailure* parameter, either the Exception is caught or will be rethrown.

- **Non-functional activity:** In case of non-functional activities — similar to non-functional actions — a single advice is generated. However, multiple middleware service invocations are generated according to the order of non-functional tasks defined in the respective activity. For example, if there are two exclusive non-functional actions A and B, an *if* statement is generated containing the corresponding middleware service invocations.

- **Non-functional Association:** For each non-functional association from a non-functional action to a task, a pointcut-advice pair is generated. The pointcut binds the method invocation join points, whereas the advice is responsible for the invocation of the middleware service. The association type is translated into the respective advice types *before*, *after* or

around. The name of the generated pointcut is the name of the action concatenated by the name of the task.

- **Interface Operation:** An operation which is part of a component's interface represents the entry point. For web services as a concrete component-based technique, an interface operation can be found in its WSDL description. This interface operation usually invokes a set of further operations implementing subsets of the operation's functionality. To store context data for one and the same invocation, e.g., for passing values from one middleware service to another, an instance of an aspect is bound to the interface operations. This is realized by first generating a pointcut named *component_execution* selecting all interface operations and secondly using the *percflow(component_execution())* keyword in the aspect definition to bind an aspect instance to the *component_execution* pointcut. This means whenever the join point selected by *component_execution* is executed, a new aspect instance will be created. Each aspect instance manages its own set of attributes containing the context data for the current web service invocation.

A model-to-text (m2t) component generates the AspectJ code for enforcing the modeled non-functional concerns

Running Example

Listing 3.10 shows the aspect generated from the mapping model presented in Figure 3.47. There are several pointcut definitions specifying a query for a set of join points. The *component_execution()* pointcut (Line 5-8) selects all web service operations of the *FlightReservationService*. It is used for binding the lifecycle of one aspect instance to the invocation of one such method. This is accomplished by using the *percflow(component_execution())* definition (Line 1). The aspect defines two instance variables *cachingservice* and *loggingservice* (Line 2 and 3) containing the instance of the respective middleware service. The *log_isReservationPossible()* (Line 10-13) and *cachingactivity_isReservationPossible()* (Line 15-17) pointcuts represent the two non-functional associations. They both point to the *isCreditCardValid* method invoked in the context of *isReservationPossible*. The advice definitions implement the invocation of the middleware services. The first advice (Line 19-23) is executed after the join points captured by *log_isReservationPossible*. It retrieves the intercepted method object and passes it to the *log* method of the logging service. The second advice (Line 25-36) is executed around the join points it is attached to and implements the process logic defined in the *Cachingactivity*. The advice executes the *readFromCache* method of the caching service and if nothing is cached, it will execute the join point (by invoking *proceed*, Line 30) and write the result to the cache.

```
1  public aspect NFA percflow(component_execution()) {
2    CachingService cachingservice = new CachingService();
3    LoggingService loggingservice = new LoggingService();
4
5    public pointcut component_execution():
6      execution(* de.bs.ws.mytravelagency.MyFlightReservationService.bookFlight(..)) ||
7      execution(* de.bs.ws.mytravelagency.MyFlightReservationService.searchBestFlight(..)) ||
8      execution(* de.bs.ws.mytravelagency.MyFlightReservationService.searchFlights(..));
9
10   public pointcut log_isReservationPossible():
11     execution(* de.bs.ws.mytravelagency.db.CreditCardProviderAccess.isCreditCardValid(..))
12     && cflowbelow(execution(* de.bs.ws.mytravelagency.MyFlightReservationService.isReservationPossible(..)))
13     && !cflowbelow(execution(* LoggingService.*(..)));
14
15   public pointcut cachingactivity_isReservationPossible():
16     ...
17     && !cflowbelow(execution(* CachingService.*(..)));
18
19   after(): log_isReservationPossible(){
20     try {
21       loggingservice.log(thisJoinPoint.getSignature().getMethod());
22     }catch ...
23   }
24
25   Object around(): readfromcacheactivity_isReservationPossible(){
26     try {
27       ...
28       returnValue = cachingservice.readFromCache(thisJoinPoint.getSignature().getMethod(), thisJoinPoint.getArgs
             ());
29       if (returnValue == null) {
30         returnValue = proceed();
31         cachingservice.writeToCache(
32           thisJoinPoint.getSignature().getMethod(), thisJoinPoint.getArgs(), returnValue);
33       } else {}
34       return returnValue;
35     }catch ...
36   }
37 }
```

Listing 3.10: AspectJ Code for Implementing Caching for the FlightReservationService

3.7 Conclusion

In this chapter, the NFComp framework has been presented and explained in detail. NFComp is a model-driven approach for the composition of non-functional concerns in component-based software applications. It defines four specification and two realization phases. In the white box view, an additional code-to-BPMN generation phase is introduced.

In the specification phases, a behavioral model is created which introduces the notion of non-functional actions representing non-functional behavior, such as algorithms for encryption. These actions can be composed; i.e., the execution order between actions can be defined as

well as the mapping from actions or action compositions to functional components. The feature interaction problem, which occurs whenever a set of features must be composed to a working software, is also addressed by NFComp but specifically for non-functional actions. NFComp allows modeling of action interdependencies at design time. These interdependencies can be used to validate the model to discover and resolve conflicts in the composition. Furthermore, a guided modeling procedure has been introduced, supporting the modeler in creating only valid action compositions.

In the realization phases the modeled behavior is automatically transformed into executable code. No further manual step is needed; only a set of middleware services implementing the modeled non-functional actions is presumed. The enforcement of the composition logic and the weaving of this logic into the software components is achieved by NFComp's proxy-based approach. The composition logic is completely decoupled from the software component and thus does not depend on any of the underlying platforms used for the implementation of the software components, e.g., web services. In the white box view, a more tightly coupled solution has been chosen. Instead of a proxy, a set of aspects is generated which are woven with the web service or even internal implementation classes. These aspects do not contain any middleware functionality but invoke the respective middleware services in the given order.

NFComp defines an abstract generic and a specific executable part. The generic framework can be used for all kinds of component-based software but lacks necessary details needed for its instantiation. However, the generic framework is complemented by a concrete instantiation for web services. In this, the different views on web services have been taken into account: the black box, gray box and white box view. For each view the different phases have been described, and the differences and extensions for the respective view have been outlined. A running example has been used to clarify the presented concepts. In contrast to the other view, the white box approach can be applied not only to web services but to component-based Java software in general.

In summary, NFComp is a platform-independent and model-driven approach for composing multiple non-functional concerns in terms of fine-granular actions. The model is transformed into executable code artifacts which are responsible for enforcing the modeled behavior.

Chapter 4

Evaluation

4.1 Introduction

This chapter covers the evaluation of the NFComp approach. Since the NFComp approach focuses on modeling, it can be seen as a kind of modeling method. Siau and Rossi [84] investigated different evaluation and comparison techniques for modeling methods. They identified three categories: feature comparison, theoretical and conceptual investigation, and empirical evaluation. With feature comparison, a list of features is selected in an objective manner. For these features, a check is done to determine which of them the approach at hand actually supports. Theoretical and conceptual investigation can be performed as metamodel evaluation, metrics analysis, paradigmatic analysis, contingency identification, ontological evaluation or cognitive evaluation [84]. Empirical evaluation includes surveys, laboratory experiments, field experiments, case studies, action research and verbal protocols [84].

In this thesis, several of the mentioned techniques are applied in order to achieve a comprehensive assessment. Firstly, an evaluation of NFComp based on the requirements identified in Chapter 2 is conducted. The lack of requirement support in state-of-the art industry and academic approaches is one of the main motivations for inventing NFComp. It should, hence, be able to fulfill the requirements to a satisfiable extent. Secondly, an evaluation is conducted based on high-level criteria commonly used to asses software engineering approaches in general. This involves, among others, a metamodel evaluation based on metrics from the conceptual investigation field. Theoretical and conceptional investigation based on the metamodel is a helpful technique because "it is purely based on the characteristics modeled in the metamodel" [84]. Thirdly, a case study (part of the empirical evaluation field) is introduced. Case studies can be used to investigate how or why (Yin [101]) a methodology should be used. They help to evaluate the instantiation of an abstract methodology in concrete scenarios. However, the selection of the scenarios should be performed as objectively as possible. To achieve this objectivity, a well-known and commonly used example scenario is applied for this thesis. Furthermore, in order to get an impression of how NFComp performs compared to other approaches, it is shown

how related approaches would implement the very same scenario. Finally, for each phase in the NFComp approach, a feature comparison is conducted. Parts of this chapter have been published in [79, 77].

4.2 Revisiting the Requirements

In this section, NComp is analyzed against the requirements identified in Section 2.5. In Section 2.6 related work has already been analyzed against these requirements revealing that there are weaknesses, essentially, in supporting requirements S5: NFA Execution Order Specification, S7: NFA Control Flow Specification, S8: NFA Data Flow Specification, as also in E5: Transparent Weaving with distributed WS and E6: Independence of Programming Language. A description on how NFComp supports the requirements follows, classified into specification and realization requirements.

4.2.1 Specification Requirements

The *non-functional requirements specification* (Requirement S1) is directly related to the requirements specification phase. In this phase, a requirements engineer defines all types of non-functional requirements in the form of informal attributes grouped by concern. The requirements may not, however, be associated with the target subject of a process, web service or operation. In order to support target subjects for requirements, the requirements engineer must also do such a mapping in her own mapping phase. Since NFComp focuses more on the non-functional actions than on requirements, this mapping has been omitted for the sake of simplicity. However, NFComp is open for such an extension as well as for the use of another more comprehensive requirements model such as the NFR Framework [18]. The dependencies between requirements and action model are minimal. For an alternative requirements model, it must only be possible for the specified requirements to be referenced from the NFComp model. This can be achieved for example by the use of URIs for all elements in the requirements model.

The *non-functional actions specification* (S2) is at the heart of NFComp. It is also directly associated with its own phase: the action definition phase. In this phase, all kinds of actions can be modeled as an abstraction of non-functional behavior realized as a software component or middleware service or any other modular software construct. An action defines different types of properties. Particular properties are used to specify the composition behavior of actions (e.g., *impact type* and *target* or *direction*), whereas others focus on the configuration of the runtime components realizing these actions (middleware mapping phase). New action models can be created and existing ones can be reused by importing them. This allows the building of a non-functional action libraries which can be used for several projects.

The *web service subjects specification* requirement (S3) directs that there be fine-grained subjects for actions such as operations, input, output, faults or the service itself. This requirement is directly supported in the action to service mapping phase. In this phase, the WSDL

of a service can be imported, which produces a graphical representation of the service and its operations. The service and its operations can be associated with a non-functional action. This is visualized with connection lines. The input, output and fault parts can be specified using the corresponding association types, which are represented by the respective symbols for line decorators.

The intention of the *NFA interdependency specification* requirement (S4) is to detect inconsistencies in the composition of actions at design time by specifying interdependencies for NFAs. NFComp adopts an existing set of interdependency types defined by Sanen et al. [76] and complements it with the *precedes* interdependency for defining ordering constraints between actions. Another additional interdependency is *inverse*, which is used to relate an action with inverse behavior to another action. NFComp offers a validation mechanism and a guided modeling procedure in order to create conflict-free (with respect to the constraints that these interdependencies impose) action compositions.

The *NFA execution order specification* requirement (S5) defines that it be possible to specify the execution order of actions depending on a concrete scenario; i.e., there should be not only a default execution order of actions but also one which is specific to a particular subject, for example a certain web service. Moreover, it should be possible to define dynamic work flow based on the execution context. These requirements are fully supported by NFComp's non-functional activity concept (covered in the action composition phase) which encapsulates an execution order of non-functional actions in form of BPMN processes. Such a process may also contain gateways allowing the definition of dynamic ordering based on conditions such as specific contents in an intercepted message or variable values in an intercepted process.

Requirement S6, *composite web service subjects specification*, is split into three sub requirements: S6.1 *control flow specification*, S6.2 *data flow specification* and S6.3 *functional specification*. NFComp supports S6.1 by allowing the association of a non-functional action with a process task. This is the node-centric approach; the transition-centric one is not supported. S6.2 is not supported by NFComp. It is only allowed to associate an action with tasks, not with data items. S6.3 is supported by associating non-functional actions with tasks using the *in*, *out* and *fault* type of associations. This enables the activation of actions during the messaging process of the composite web service.

Requirement S7 prescribes that *control flow of composite NFAs* can be specified. This is covered by NFComp by the concept of composite NFAs which can be modeled similarly to non-functional activities in BPMN2. The composite action definition is translated into interdependencies which then can be validated in order to make sure that the action composition and mapping conforms to the control flow of composite actions.

NFA data flow specification (S8) defines data flow between actions. This allows transportation of data input/output between actions. This requirement is supported in NFComp by specifying data items and -associations between actions. The container for such data flow must be a non-functional activity or a composite non-functional action.

In summary, NFComp supports all specification requirements except S6.2. An overview is given in Table 4.1.

ID	Requirement	NFComp
S1	Non-functional Requirements Specification	+
S2	Non-functional Actions Specification	+
S3	Web Service Subjects Specification	+
S4	NFA Interdependency Specification	+
S5	NFA Execution Order Specification	+
S6	Composite WS Subjects Specification	+
S6.1	Control Flow Specification	+
S6.2	Data Flow Specification	-
S6.3	Functional Specification	+
S7	NFA Control Flow Specification	+
S8	NFA Data Flow Specification	+

Table 4.1: Specification Requirements Support by NFComp

4.2.2 Realization and Enforcement Requirements

The first realization requirement *separation of concerns* (E1) is well supported by NFComp. The architecture used for realizing the model heavily relies on the separation of concerns principle. Firstly, the code for realizing the non-functional actions is separated. For each such action, a middleware component or class is assumed that implements the necessary functionality. However, in the end it is up to the NFComp user how to decompose her actions into middleware services. Theoretically, one middleware component could realize multiple NFAs. However, the functional and non-functional components are strictly separated. In fact, this allows introducing non-functional actions into existing services without changing the service itself at all. The middleware components are integrated into the services using a proxy approach. Thus, a proxy component is used which encapsulates the composition logic that has been modeled. In summary there are three types of components: functional components (e.g., business services), non-functional components (e.g., middleware services) and components implementing the integration and composition of the former component types (for instance, an enterprise service bus with a set of proxies).

The second requirement, E2, *weaving of FCs and NFCs*, is — as stated above — realized via a proxy component. This proxy component is set in front of a service in order to intercept all incoming and outgoing messages. However, such a proxy component can additionally be installed in front of the consuming application. This strategy can also be used to address non-functional requirements at the consumer side. When the proxy observes a message that is delivered to a service or process task which is associated with a particular NFA defined by the non-functional model, it calls the respective middleware service mapped to the actions. This is the way NFComp realizes the weaving of FCs and NFCs. The strategy, thereby, is runtime

weaving allowing the deployment of new actions while the service is already being executed. In the white box view, NFComp relies on static weaving based on bytecode modification. This is due to the use of AspectJ aspects instead of proxies.

Requirement E3 called *quantification* defines that it should be possible to quantify over services, service operations, process tasks and so on. This means a single action can be applied to multiple functional subjects at once. Often, dedicated query languages are used for this purpose. However, NFComp uses a graphical model to select the functional subjects. One and the same action can be associated with different services or tasks. This strategy is rather explicit; the user must select the subjects directly instead of using a particular kind of query language. This has advantages and disadvantages. The advantage is that the user directly sees which functional points are associated with a non-functional action. Moreover, if the functional model changes, this change will also affect the non-functional specification; e.g., if a task is removed from a process definition, the non-functional association will also be removed. Using a query language, however, is more powerful, for instance a set of dozens of services can be woven with the same action at once without too much effort. In NFComp, however, the same number of non-functional associations must be modeled manually. On the realization level, NFComp also supplies a certain grade of quantification. An additional layer of indirection exists between proxy components and mediators. For each non-functional association, a dedicated proxy sequence is generated. Thus, one and the same mediator sequence can be used for different service subjects. However, this is — similar to the approach at the code level — a kind of explicit quantification.

Requirement E4, *superimposition*, allows the support of superimposing NFAs, i.e., multiple NFAs — and thus middleware services — applying to the same functional point. A particular, previously specified execution order should apply in this case. NFComp supports this requirement by generating a proxy configuration out of the non-functional model. This proxy configuration realizes the control flow specified by non-functional activities through the invocation of mediators in the respective order. For each web service a specific control flow of mediators can be used. Also the white box view approach allows the superimposition of multiple NFAs. This is realized in the implementation of the AspectJ advice which invokes several middleware services in the same order as modeled in the mapping phase.

Transparent weaving with distributed web services (Requirement E5) is a requirement which is supported by NFComp but with restrictions. In general, NFComp supports the weaving of NFAs into distributed web services. However, for particular NFCs multiple proxies or multiple middleware services of the same type must be used. For example, reliability and security concerns require complete end-to-end support. If the providers of different distributed web services were using the same shared middleware service for security, respectively reliability, there could be an insecure or unreliable transport path from the proxy to the middleware web service (it can either be hosted by Provider A or Provider B or even by a third party). Hence, it is not completely transparent for the user whether web services are distributed over the Internet or not.

Furthermore, it is not realistic for a service provider to allow other providers the deployment of new middleware services to their business web services, although it is technically feasible as long as both are using NFComp with the same ESB infrastructure.

Independence of programming language (Requirement E6) is a main principle followed by NFComp. The code for the non-functional actions is completely independent of that of the services by the use of the decoupled proxy component. It is also possible to decouple the proxy component from the middleware services. This can be achieved by implementing the middleware services as web services. The programming language used for the implementation of local mediators is restricted to the same programming language as the one used for the proxy. In the white box view approach both, the code for action composition (AspectJ aspects) and the middleware services implemented as Java classes depend on the programming language used for the web service implementation. However, this is a general drawback of the white box approach. If complete independence of the web service implementation must be achieved, either the black or gray box view approach should be used.

The strengths of the NFComp approach lie in the well-defined, role-based development process that has been defined in order to cope with the inherent complexity of NFC composition. Through the NFComp approach, complex static and dynamic workflows can be specified in the form of non-functional activities. To ease the complexity of this task, additional domain knowledge about interdependencies and properties of actions has been captured during the action definition phase. Moreover, the specification conforms to a metamodel and hence can be used for code generation. The generated code supports the requirements with restrictions for Requirement E3 (*quantification*) and E5 (*integration of NFCs with distributed WS*). In summary, almost all realization requirements are fully supported by NFComp. Table 4.2 outlines the evaluation results.

ID	**Requirement**	NFComp
E1	Separation of Concerns	+
E2	Weaving of FCs and NFCs	+
E3	Quantification	(+)
E4	Superimposition	+
E5	Transparent Weaving With Distributed WS	(+)
E6	Independence of Programming Language	+

Table 4.2: Enforcement Requirements Support by NFComp

4.3 Criteria-Based Evaluation

The last section showed that the NFComp approach supports almost all of the requirements identified in Chapter 2. This was one of the main goals for inventing this novel and holistic approach. However, there are also high-level criteria for evaluating the quality of engineering approaches in general. For example, Parastoo [70] presents a set of evaluation criteria for model-

driven engineering approaches. This set of criteria has been adopted and extended by the most important principles for modular design. The criteria are used below in order to evaluate the overall quality of the NFComp approach. The following criteria have been adopted for the evaluation: Scope of Application, Specification Complexity, Standards Compliance, Extensibility, Correctness of the Specification, Expressiveness of the Composition Language, Completeness, Separation of Concerns, Support for Multiple Users, and Support for Multiple Roles, Multiple Views and Multiple Execution Environments.

4.3.1 Scope of Application

The scope of application describes the different areas to which an approach can be applied. The wider the scope of application, the more problems can be addressed by such an approach and the better the chance of this approach for being adopted in order to solve a particular problem. One possibility to increase the scope of application is to provide a high level of abstractness and generality.

The scope of application in NFComp can generally be investigated for functional and non-functional concerns. Regarding its applicability for functional concerns, NFComp is two-fold: It provides an abstract framework that can be applied to component-based software applications in general, and a concrete, instantiable one for web-service-based applications. If the underlying software system to be enhanced with non-functional concerns is based on web services, the concrete, web-service-specific NFComp approach can be used. If the software system is component-based but is not using web services, the abstract approach can be used as a framework to build a new concrete application of NFComp for the specific technology. An example for such an alternative, component-based system could be REST-based services or web applications in general. The differentiation of abstract and concrete parts in NFComp assures the widest possible scope of application.

Regarding non-functional concerns, NFComp is a generic framework applicable to all types of NFCs. This distinguishes NFComp from many other approaches which are applied only to specific non-functional concerns such as security (for example, Sec-MoSC [88]), or monitoring (for example, Kallel et al. [49]). Furthermore, it is easy to add new, unanticipated NFCs, whereas in other generative approaches there is a higher implementation effort (for instance, Sec-MoSC [88] or AO4BPEL [12]). The main reason is that the generated code of NFComp can deal with all kinds of middleware as long as implemented as web service, whereas most other approaches require at least a modification of the generator to introduce a new NFC. For example, to introduce caching in AO4BPEL, the deployment descriptor must be extended by new elements and a new aspect template for the aspect generator must be written to generate aspects which call the caching middleware web service.

4.3.2 Specification Complexity

The specification complexity defines how difficult it is to learn and make use of the specification language provided by an approach. The higher the complexity, the higher the effort to apply the approach to a given problem. There are two aspects to be analyzed with respect to this regard: the complexity compared to other graphical modeling languages and the complexity compared to textual specification languages.

Rossi and Brinkkemper [74] propose a systematic approach for measuring properties of methods to analyze their descriptive capabilities. The goal of this approach is to measure how complex it is to understand and learn a method or technique, and how complex the internal structure of a model resulting from applying a methodology is. The former strongly depends on the number of concepts used, whereas the latter strongly depends on properties describing these concepts. The metrics defined by Rossi and Brinkkemper can be separated into independent and aggregate metrics. The function $n(A)$ counts the number of elements in a set A. A description of the most important metrics (same metrics as used in Indulska et al. [47]) is given in the following:

1. $n(O_T)$ counts the number of individual object types used for one technique.
2. $n(R_T)$ counts the number of relationships used for one technique.
3. $n(P_T)$ counts the number of properties for one technique.
4. $\overline{P}_O(M_T)$ counts the average number of properties per object.
5. $\overline{P}_R(M_T)$ counts the average number of properties per relationship.
6. $C'(M_T) = \sqrt{n(O_T)^2 + n(R_T)^2 + n(P_T)^2}$ represents the overall complexity by calculating the length of the vector $(n(O_T), n(R_T), n(P_T))$.

Method T	$n(O_T)$	$n(R_T)$	$n(P_T)$	$\overline{P}_O(M_T)$	$\overline{P}_R(M_T)$	$C'(M_T)$
$NFComp_{ReqModel}$	2	1	3	0.33	0	3.74
$NFComp_{ActionDefModel}$	1	1	8	6	2	8.12
$NFComp_{ActionCompModel}$	10	4	5	0.4	0.25	11.87
$NFComp_{BlackboxMapping}$	5	1	6	1	1	7.87
$NFComp_{GrayboxMapping}$	6	1	7	1	1	9.27
$NFComp_{WhiteboxMapping}$	4	1	5	1	1	6.48
$NFComp_{All}$	20	7	24	0.95	1.5	32.02
$BPMN$ [47]	57	6	74	1.19	1.33	93.60
$UML_{ActivityDiagram}$ [47]	8	5	6	0.75	0.2	11.18

Table 4.3: Metrics for Graphically Represented Metamodel Concepts

Table 4.3 shows the resulting metrics for the NFComp modeling approach, which is compared to other modeling methods such as UML and BPMN. The metrics are listed separately

for each diagram type provided by NFComp and in addition there is an aggregation for all NFComp diagrams called $NFComp_{All}$. Concepts which are used in multiple diagram types are counted once. Hence, the aggregation of the metrics in $NFComp_{All}$ is not equal to the sum of the individual diagram types. The particular metrics have been collected as follows: For $n(O_T)$ the object types with a graphical representation in the respective diagram have been counted, e.g., non-functional concern and non-functional attribute for the requirements model. For $n(R_T)$ the graphically represented relationships have been counted, e.g., control flow, data flow, default control flow and the containment relationship between process elements (such as non-functional tasks) in the case of the action composition diagram. The graphically represented properties $n(P_T)$ for this diagram type are the *name* properties of non-functional activity, composite non-functional action, non-functional task and data item, which are all object type properties, and the *condition* property for sequence flows, which is a relationship property. The metric $\overline{P}_O(M_T)$ for this diagram type is 4/10 because 4 properties are associated with object types and 10 object types exist. $\overline{P}_R(M_T)$ is 1/4 because there is 1 property associated with relationships and 4 relationships exist.

The resulting metrics show the simplicity of the NFComp modeling method. In the case where each diagram type is regarded individually, NFComp is comparable to UML Activity Diagrams. Since there is a strict separation of concerns, i.e., different types of diagrams are modeled by different persons, the methodology allows dealing with each diagram individually. However, even if one person plays multiple roles and must understand how to use the whole set of diagrams, the metrics show that NFComp is obviously less complex than BPMN.

Compared to textual specification languages such as XML-based ones, the graphical model provided by NFComp is easier to read and to understand. Firstly, the use of visual graphs as representation for processes or behavior of programs has been widely adopted, for example in BPMN or UML Activity Diagrams. Secondly, the visualization of the mapping between actions and services represents a relationship. Relationships between entities are also visualized with connections in Entity Relationship Models [15] or UML Class Diagrams. They are much easier to understand than textual mapping approaches such as Hibernate[1] mapping or JPA[2] annotations. This is why many model-driven frameworks such as AndroMDA[3] and OAW/Fornax[4] generate this mapping out of a graphical representation, e.g., Class Diagrams with associations.

4.3.3 Standards Compliance

The standards compliance criterion defines how strongly an approach relies on open standards and how many proprietary components are used. The advantage of high standards compliance are, among others:

[1] http://www.hibernate.org/
[2] http://www.jcp.org/en/jsr/detail?id=220
[3] http://www.andromda.org/
[4] http://www.fornax-platform.org/

- **Ease of adoption.** Relying on standards decreases the effort for the adoption of the approach. This is because there is a greater chance of finding experienced people familiar with the standard. The already existing knowledge can be reused, and time and thus effort can be saved.

- **Portability of solution.** The portability of the solution increases when it is compliant with standards. This is because there are usually several providers offering different tools around a certain standard. This allows movement from one provider to another without the necessity of rewriting the specification or code. There is no vendor lock-in at all.

NFComp abides by standards wherever possible. It uses BPMN2 for the definition of non-functional activities and thus the composition of non-functional actions. Also, composite non-functional actions are defined by BPMN2. In order to reduce the complexity of the generic business process modeling language which also supports advanced concepts such as choreographies, only a small subset is used. Furthermore, BPMN2 is used for the modeling of composite web services in the gray box view. In the white box view, the service behavior is extracted from code and transformed into BPMN2. This makes the BPMN2 standard the core language for NFComp. NFComp has been designed with a focus on the web service domain where BPMN2 is widely known and applied and provides a graphical modeling standard. Besides, the WSDL standard for black box services and the WS-BPEL standard for executable processes has also been used.

Nonetheless, there are also proprietary language constructs. For the graphical visualization of services from the black box view, a self-invented, simplified notation is used and for the mapping different association notations have also been introduced. These concepts are rather obvious and easy to learn (cf. Section 4.3.2). The same applies to the requirements notation. The reason for the use of proprietary constructs in this case is that no suitable standards for this purpose existed at the time when this thesis was written.

4.3.4 Extensibility

An approach is extensible if it can be adapted to unanticipated changes (cf. Zenger [104]). There are two aspects with respect to extensibility: firstly, how easy it is to extend the approach, and secondly, how powerful the extension mechanism is. Extensibility is an important criterion, because it is hard to foresee all possible changes that can be applied to an approach. Hence, there should be general, well-defined extension points that can be used by developers and users of the approach. In general, an extension should not require the extender to change the existing code or specification. NFComp is a model-driven approach, hence, both the extensibility of the specification and the extensibility of the generated code must be taken into account.

The extensibility of the specification depends on the extensibility of the model and its metamodel. The metamodel extensibility helps to extend the language by new concepts or to adapt the language by changing or removing particular elements. This could, for example, be in-

tended by a developer who wants to adapt the approach to new requirements which have not been anticipated before and cannot be supported with the standard approach.

The metamodel of NFComp has been specified by Eclipse Ecore, which is a MOF [65] compliant metamodel. MOF provides the concept of a *Tag* element which can be applied to all concepts of type *Element*. Each *Tag* has a *name* and a *value* property. This concept is implemented by Ecore's *EAnnotation*, offering a flexible way to annotate Ecore models with additional metadata. This annotation concept — compared to a direct change of elements in the metamodel — is a relatively lightweight way of extending the metamodel. The generator that transforms the model into code can access the annotations and generate different content based on the extensions.

Besides this lightweight way of extending a metamodel by annotations, a heavyweight extension in terms of new or modification of existing model elements is also possible. The modification of elements is always disruptive, whereas the introduction of new elements can be accomplished without changing the existing metamodel. For this purpose, a new element can extend (object-oriented inheritance semantics) an existing element which allows the usage of this new element instead of the extended one (polymorphism). The new element should be defined in a new metamodel file in order to separate extensions from the standard approach.

The second aspect of the specification extensibility is the extensibility of the model, which reflects more the user's perspective than the developer's perspective. This means, in the case of NFComp, determining how easy it is to extend an already existing model (requirements, action, action composition, mapping) by new non-functional concerns or web services. Firstly, the metamodel of NFComp is very generic in terms of the types of NFCs that can be supported. Secondly, the fine-grained file structure of the models allows the import of new actions into already existing ones or also existing ones into new ones.

The extensibility of the code generated by model-driven approaches is important, especially for those approaches which generate code skeletons that need to be extended manually later on. Model-driven approaches, such as AndroMDA or OAW/Fornax follow this strategy and generate only parts of the code. For example, the body of methods is not actually generated. Only a method head is generated, which must be overridden in a subclass. This subclass is dedicated for manual extensions and will not be overwritten by the code generator.

NFComp generates the proxy configuration with the composition logic for mediator execution according to the non-functional activity definition. The proxy configuration is completely generated; there is no need for any manual modification. It is more likely to implement new mediators to extend the set of existing mediators provided by the enterprise service bus. There are two ways of achieving this: A new middleware web service or a new local mediator can be implemented. The former way is straightforward. The new middleware functionality is implemented as a web service and mapped from a non-functional action by specifying the web service address and operation. The latter way is more complex and requires an extension of the model-to-model transformation logic. A local mediator name must be defined for the new mid-

dleware functionality, and for this name a new XML element in the proxy configuration must be generated.

In the Synapse ESB, a mediator is implemented as a Java *Class* extending the *AbstractMediator* class and overriding the abstract *mediate* method. Additionally, a new class extending the *AbstractMediatorFactory* must be written which creates the Java object out of the XML element specified in the proxy configuration. Adding this additional mediator as a Jar file to the library (called libs) folder of the Synapse installation enables the integration of the new middleware service. The white box view uses aspects for middleware service integration, but they are also completely generated. It is simple to add a new middleware service, because it suffices to implement a new Java class and map it to a non-functional action.

4.3.5 Correctness of the Specification

The correctness of the specification criterion determines whether it is possible to validate the correctness of the specification during the time of modeling. If the specification is not correct at design time, this may result in errors/problems at runtime. Early feedback about the correctness of the specification is very important. This is for example a benefit of statically typed languages over dynamically typed ones. The compiler is able to provide a type check at design time (type safety), so that the programmer will immediately see errors related to types. In dynamically typed languages, however, type errors can only be detected at runtime, which is a severe drawback because the feedback is late and hence it takes too much time to fix the error. Furthermore, there are two types of correctness: syntactical and semantic correctness. Classical compilers are usually only able to check for syntactical correctness.

The same concepts also apply to graphical modeling languages. The syntactical correctness of graphical specification languages is often assured by the tool or editor which is used to create/edit the specification. This means it is either only possible to create valid specifications, or the specification can be validated and feedback is provided in the form of validation errors. Semantic correctness goes beyond the syntactical correctness and defines the correctness of the meaning of the specification contents.

In NFComp, the syntactical correctness is assured by the editors which check that the produced model always conforms to the metamodel. A certain degree of semantic correctness is assured by the validation mechanism based on the interdependency model. This interdependency model covers particular semantic aspects of action compositions such as specific dependencies between non-functional actions. The interdependency model constrains the action composition, and NFComp allows the validation of the model against those constraints. Furthermore, a guided conflict-free composition procedure is offered.

4.3.6 Expressiveness of the Composition Language

For an objective measurement of the expressiveness of the composition language, the well-known workflow patterns can be used. The workflow patterns are commonly used to compare

the expressiveness of workflow languages or even to build new languages, e.g., YAWL (Yet Another Workflow Language, van der Aalst and Hofstede [92]). There are currently five different categories of patterns: controlflow, resource, dataflow, exception handling and presentation. The most relevant categories are the controlflow and dataflow patterns. The more patterns supported, the higher the expressiveness of the composition language, however, some of the patterns are more specific or exotic and are not as important as others. Van der Alst and ter Hofstede[5] provide a catalog of workflow patterns to evaluate the power of well-known workflow languages such as BPMN or WS-BPEL.

ID	**Pattern**	**NFComp**	**BPMN 1.0**	**Soeiro**	**Fox**	**Ortiz & Hernandez**
C1	Sequence	+	+	+	+	+
C2	Parallel Split	+	+	+	-	-
C3	Synchronization	+	+	+	-	-
C4	Exclusive Choice	+	+	-	-	-
C5	Simple Merge	+	+	-	-	-
C6	Multi-Choice	+	+	-	-	-
C7	Structured Sync. Merge	+	+	-	-	-
C8	Multi-Merge	+	+	-	-	-
C9	Structured Discriminator	-	+/-	-	-	-
C10	Arbitrary Cycles	-	+	-	-	-
C11	Implicit Termination	+	+	-	-	-
C12-15	Multiple Instances	-4	+3/-1	-4	-4	-4
C16	Deferred Choice	-	+	-	-	-
C17	Interleaved Parallel Routing	-	-	-	-	-
C18	Milestone	-	-	-	-	-
C19,20	Cancel Activity/Case	-2	+2	+1/-1	-2	-2
C21	Structured Loop	+	+	-	-	-
C22-42	Further Patterns	-21	+6/-7	-21	-21	-21
C43	Explicit Termination	+	+	-	-	-
D1	Task Data	-	+	-	-	-
D2	Block Data	-	+	-	-	-
D3-D8	Further Patterns	-6	+1/-4	-6	-6	-6
D9	Task to Task	+	+	-	-	-
D10-39	Further Patterns	-30	+14/-13	-30	-30	-30
D40	Data-Based Routing	+	+	-	-	-

Table 4.4: Workflow and Dataflow Patterns Supported by NFComp and Other Approaches

NFComp is a subset of BPMN and thus not as expressive as BPMN itself. In order to measure its expressiveness, an investigation has been made of which of the workflow patterns (more specifically controlflow and dataflow patterns) are supported by NFComp. Table 4.4 shows the results of this investigation and that of other comparable approaches. In this table, +

[5] http://www.workflowpatterns.com/

means directly supported, +/- not directly supported and - no support. The numbers right to + or - define the number of +, respectively -, for patterns shown in an aggregated form. An approach was chosen to be compared to NFComp and BPMN when it had a positive rating with respect to Execution Order Specification (Requirement S5) in the evaluation of related work in Section 2.6. The evaluation results for BPMN 1.0 have been taken from Wohed et al. [99].

The evaluation results in the table show that NFComp is less expressive than BPMN. This is natural since NFComp supports only a subset of BPMN for the specification of non-functional activities or composite non-functional actions. Nevertheless, compared to related approaches targeting at non-functional concerns, NFComp supports definitely more workflow patterns. In this area, the composition language of NFComp is superior in its expressiveness. The reason for not supporting some of the workflow patterns that BPMN supports, is that no use case has been found for this pattern for non-functional action composition. To keep the composition language simple, such patterns were intentionally not supported.

4.3.7 Completeness

The completeness of an approach can be measured as the grade of its requirements fulfillment. It is assumed that whenever there is a need for the invention of a novel approach, requirements exist that are not yet covered by any other approach. The requirements that lead to the approach should then be suitably supported.

In Section 4.2 the requirements identified in Chapter 2 have been revisited in order to evaluate NFComp against these requirements. The analysis showed that NFComp supports almost all requirements in an appropriate way. This makes NFComp a complete and holistic approach for the composition of non-functional concerns.

4.3.8 Separation of Concerns

Separation of concerns (SoC) is one of the most important principles for a clear design of software in order to cope with complexity. This principle can be applied to the specification as well as the code generated out of the specification. In specification and code it is important to separate functional from non-functional concerns in order to achieve a clear and modular design. The SoC principle has an impact on further sub-criteria: support for reuse and ease of change. The better the SoC, the higher the chance to reuse a component; e.g., if non-functional concerns are completely separated, the components implementing the NFCs can be reused in different contexts and scenarios because they are not tied to a particular functional component. Furthermore, if each concern has been clearly separated and modularized, into a component — whether functional or non-functional — a change will probably impact only this very component and will not affect others. At the code layer, it is important to separate handwritten from generated code.

NFComp focuses mainly on the separation of non-functional and functional concerns at code and specification level. In the specification, distinct models which can themselves be split

into separate files are used for the different phases. This allows for a separation of the model into different non-functional concerns, e.g., a model file for security containing only security-related actions and an extra file for the reliability-related ones.

Regarding the code level, functional concerns are assumed to be the core components of a system such as web services. Non-functional concerns are realized by extra components such as middleware services. Another component, the ESB, is responsible for the integration and composition of these middleware web services with the core components. The separation of hand-written and generated code is strict. The proxy configuration is completely generated and middleware services are assumed to be written by hand (at least they are not generated in NFComp). Still, it is theoretically possible to replace particular generated files by handwritten ones. This is enabled by the strong separation of configuration files; for example, a mediator sequence can be replaced by a manually written one without affecting the others. However, the replaced file will probably be overwritten in the next generation cycle.

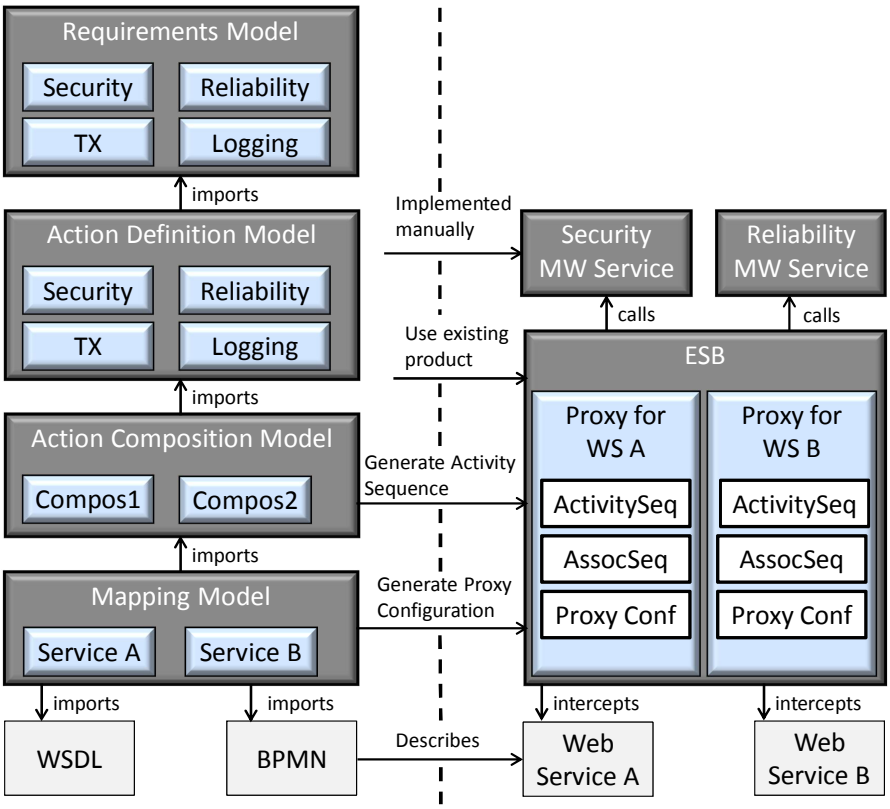

Figure 4.1: Separation of Concerns in NFComp

Figure 4.1 shows the different components in NFComp which result from this clear separation of concerns. On the left-hand side, the specification components are depicted, whereas on the right-hand side the different runtime components are shown. As can be seen in the figure, there are four different models created in the respective phases. Each such model can be decomposed into individual files, for example, per concern. One such model can import another

one, and functional models such as WSDL or BPMN description can also be imported. Middleware services are implemented manually and can be invoked from an ESB which is an already existing product. The configuration of the ESB is via XML files. Those files are decomposed into proxy definition files (containing the proxy-related stuff such as the link to the target web service and the invocation of mediator sequences), association sequences (generated per non-functional association pointing to activity sequences) and activity sequences implementing the control flow defined by non-functional activities.

4.3.8.1 Support for Reuse

The support for reuse criterion defines how much of the artifacts created during the methodology can be reused in different scenarios. High reuse allows the avoidance of duplication of specification contents. Duplications in the specification make it hard to maintain; e.g., when the content is changed, the duplicated content also must be changed accordingly. There are specification artifacts as well as implementation artifacts that should be reusable.

In NFComp, the models created in the different phases can be reused in different contexts. In particular, the reuse of the requirements model and action definition is high because the same requirements/actions could make sense for different types of services. Also, building a library of non-functional actions is encouraged in order to collect as much knowledge as possible across different projects. The actions library can also be used as a starting point for a new project and would give an overview about what actually exists. This helps to identify appropriate actions in different scenarios. Also the action composition model can be reused whenever the same or similar collections of actions need to be mapped to a service. The service mapping will change from project to project since other, new services will be implemented and thus a new mapping must be defined.

The reuse of implementation artifacts is achieved by the strict separation of non-functional concerns and functional concerns at the code level also. Middleware services implementing non-functional concerns can be reused across different web services. Only new proxy configurations need to be generated in order to integrate middleware services into further web services.

4.3.8.2 Ease of Change

The ease of change criterion defines how easy it is to change the specification, e.g., when new requirements arise. The ease of change is influenced, in addition to SoC, by the coherence of components; i.e., when a certain concern must be changed, the number of components/parts that need to be changed accordingly. The higher the coherence, the better the chance that the change is localized in a single component or part.

In NFComp, changes in the specification will automatically result in changes in the code. This is important because otherwise both code and specification must be kept in sync manually. Manually managing both code and specification would result in great effort and high error-proneness. Changes in the NFComp model are easy to accomplish since there are different

models for each phase. Furthermore, it is possible to separate a particular model into different files. This enables localization of the change to a single file. For example, if a new service needs to be mapped with non-functional actions, a new mapping file can be created without affecting any other existing mappings. If the same actions are required for the service as for other services, they can be reused without a change. If the composition logic is different, only the composition logic needs to be adapted, and so forth.

4.3.9 Support for Multiple Users

The support for multiple users criterion defines whether it is possible for multiple user to apply the approach in parallel, i.e., working collaboratively on the same project. This is important because otherwise the working process is sequential and thus ineffective.

The process in NFComp is split into different phases. These phases are sequential and must be processed one after another. However, the model created in one phase can be split into multiple files, allowing multiple users such as non-functional domain experts to work in parallel. It is for example possible to work in parallel on security and reliability concerns until the action composition phase starts. In this phase all types of actions need to be defined in order to specify the composition of actions from different domains.

4.3.10 Support for Multiple Roles

The criterion support for multiple roles aims at the separation of the users of an approach into different roles with different responsibilities. This allows reduction of complexity in applying the approach, because a user in a certain role is not required to learn how to use the whole approach but can focus on a logically separate aspect.

NFComp provides a roles concept and restricts the different phases to the particular roles involved. However, theoretically a user can also act in multiple roles. The requirements engineer is responsible for the requirements specification. The non-functional domain expert creates the action definition model. In the action composition phase, the service provider decides which combination of actions she wants to use and can model the non-functional activities. Composite non-functional actions are rather modeled by the non-functional domain experts. NFComp already tries to collect the most important information with respect to the action composability in the action definition phase. Depending on the quality of this model, the service provider may be able to compose the actions himself. Otherwise, she can do it in collaboration with non-functional domain experts. In the mapping phase, the service provider is responsible for the mapping of activities/actions to services.

4.3.11 Support for Multiple Views

There are different views on web services: black, gray and white box. The different views on services allow different levels of non-functional concern integration but their applicability also

depends on the nature of the service. The black box view assumes web services to be black boxes in which only the interface is visible, whether they are composite or atomic. If services are composite in the sense that a service-oriented composition language — such as BPMN2 — has been used to compose the web service out of other partner web services, the gray box view is suitable to achieve a more advanced integration of concerns. For example, it is also possible to integrate composite non-functional actions for monitoring, transactions and so on. The white box view even allows a view into atomic web services or components written in Java. This makes it possible to integrate composite non-functional actions with internal code artifacts. Consequently, in order to support the different kinds of web services, it is important to support the different views. Table 4.5 summarizes when to use which view. All three views are supported by NFComp. It is also possible to combine the black and gray box or black and white box views. Non-functional actions can be mapped to the interface of the service by the classical black box approach and the internal parts of the service by the gray or white box view. This makes NFComp applicable to all types of web services.

Web Service	**Non-functional Actions**	**View**
Atomic	Atomic	Black Box
Composite (BPMN-based)	Atomic	Gray Box
Atomic	Composite	White Box
Composite (BPMN-based)	Composite	Gray Box

Table 4.5: When to Use Which View for Web Services

4.3.12 Support for Multiple Execution Environments

The support for multiple execution environments validates whether the approach can be realized in different settings for several platforms. To be suitable for multiple execution environments, an approach should be as platform-independent as possible. Furthermore, it must be generic enough to be applicable to more than one target environment.

In general, NFComp provides a platform-independent model (only parts of the middleware mappings are platform-specific). The supported target environments are web service-based architectures using an ESB as a middleware component, which is loosely coupled to the web service to be enhanced by middleware functionality. This environment supports various use cases and can be applied to atomic as well as composite web services. Furthermore, a tight integration of middleware functionality into the web service's implementation is supported via aspect generation. In this case the solution is, on one hand, specific to the underlying programming platform used for the web service implementation, and, on the other hand, can be applied not only to web services but Java code in general.

Whenever support for a new execution environment is needed, the generator must be extended and new transformation code must be implemented. Additionally, new mapping strategies may also be required. The class *MiddlewareMapping* in the metamodel reflects the differ-

ent mapping strategies. It can be extended by further, more specific mappings. An extension has already been implemented for the mapping in the white box approach. The white box approach proves the support for multiple execution environments in NFComp. It supports a completely different runtime environment based on AspectJ aspects and is not tied to web services. This shows that NFComp is open for new unintended execution environments (compare Section 4.3.4). The ESB/proxy solution is generic enough to be applied to different types of web services independently of the programming platform being used. However, there are also some general limitations with proxies which cannot always be applied due to company policies, or they may cause performance reductions (c.f. Section 4.5.2). This would motivate alternative execution environments. For the AspectJ-based generation approach, alternative aspect technology can be used, e.g., AspectC++[6] or phpAspect[7]. For this purpose a new generator must be implemented and, as discussed above, it could be necessary to extend the metamodel.

4.3.13 Summary

NFComp, as a holistic approach for the composition of non-functional concerns, supports all of the identified criteria. The results of the evaluation have been summarized in Table 4.6.

Criterion	**NFComp**
Scope of Application	Component-based software and web services
Specification Complexity	Simple graphical notations in different models
Standards Compliance	BPMN2, WS-BPEL, WSDL and proprietary mapping
Extensibility	Metamodel, model and code
Correctness of the Specification	Syntactic and semantic (interdependencies)
Expressiveness of the Composition Language	Lower than BPMN2, more powerful than related approaches
Completeness	All identified requirements supported
Separation of Concerns	Strong separation in specification and realization
Support for Reuse	Requirements-, action-, (composition)-model
Ease of Change	High, through strict separation of concerns
Support for Multiple Users	Per phase, through model/file separation
Support for Multiple Roles	Different roles involved in different phases
Support for Multiple Views	Black, gray and white box
Support for Multiple Exec. Environments	Proxy-based, AspectJ, others by implementing new generators

Table 4.6: Overview of Criteria-Based Evaluation of NFComp

[6] http://www.aspectc.org/
[7] http://code.google.com/p/phpaspect/

4.4 Case Study and Feature Comparison

In this section, the NFComp approach is applied to the purchase order scenario. This scenario has already been introduced as a running example for the gray box view in the previous chapter. However, this instantiation is done in more detail and using all three views: black, gray and white box.

In addition to implementing the scenario via NFComp it is also shown how related approaches can be used to implement the very same scenario comparing them directly to NFComp. This is done in a phase-by-phase manner. In each phase, different sets of related approaches are selected for comparison. The reason for this approach is that, as already shown in Chapter 2, most approaches are not able to cover all phases supported by NFComp. Two criteria were used for selecting related work, (a) the relatedness of an approach to the particular phase and (b), the results in the requirements-based evaluation in Section 2.6. Approaches from academia as well as industry standards have been regarded. At the end of each phase, a feature comparison is conducted. In this feature comparison, feature points are used to achieve a measurable comparison between the different approaches. Features rated with *yes* receive 1 feature point, *(yes)* receive 0.5 feature points and *no* receive 0 feature points. Ratings for choices other than *yes*, *(yes)* or *no* are explained explicitly when the respective feature is introduced. Notice that feature points do not represent the time that is required to implement this feature, but instead define how many features are supported.

In this case study, a typical purchase order scenario is presented. The rationale behind this decision is that the purchase order is a well-known real world scenario which is also frequently used as an example in the context of web services, for instance in the WS-BPEL [2] and BPMN2 [67] specification. The purchase order is a business process provided by a particular company enabling the ordering of goods. This process involves different partners and comprises distinct atomic activities to be performed. A web service is an appropriate means to make this process available to customers, because it makes use of open standards and is platform independent. The web service itself offers no graphical user interface to its customers but can be consumed by other technical applications such as web services, web applications, rich client applications, mobile applications and so on. This case study considers all views of web services. Although the purchase order scenario is a composite web service rather than an atomic one, it can be regarded from the black box view and the gray box view. The white box view can be applied when assuming the purchase order process to be a pure Java implementation. This Java code can then be reverse-engineered into a behavioral BPMN model to start with.

In the purchase order scenario regarded from the black box view, the WSDL interface description is the only valuable technical information that is available for composing web services with non-functional concerns. A reduced subset of the WSDL interface (some port types and operations have been omitted) is shown in Figure 4.2 depicting two port types: one for external partners of the service (in this case the partner who implements invoice processing) and one for consumers of the service. The *invoiceCallbackPT* defines a single one-way operation

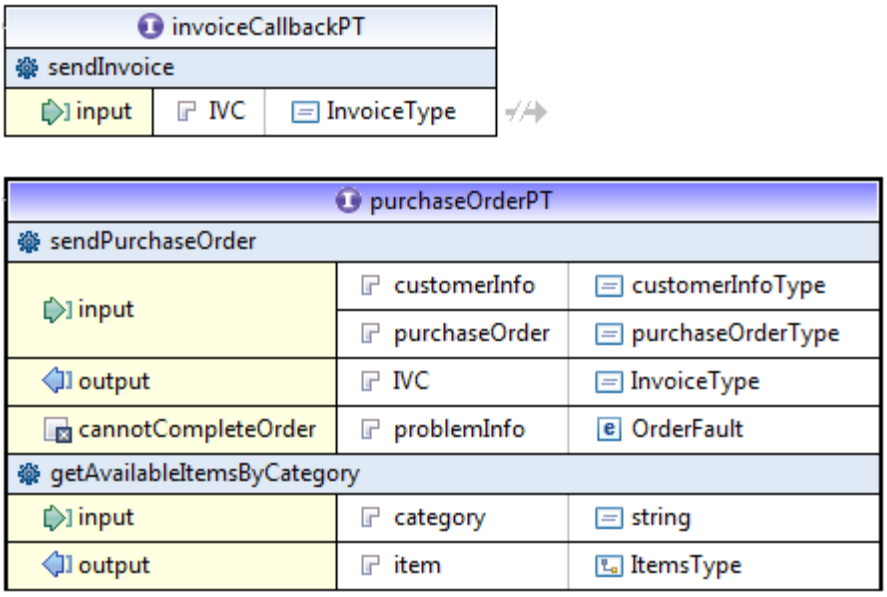

Figure 4.2: WSDL of the Purchase Order Service

named *sendInvoice*, and the *purchaseOrderPT* defines a *sendPurchaseOrder* operation and a *getAvailableItemsByCategory* operation. The *sendInvoice* operation facilitates the generation of the invoice by an external partner of the service provider and adds this information during the execution processing of the *sendPurchase* order operation. The *getAvailableItemsByCategory* operation allows users to query a list of the available items they can purchase, filtered by category. The operation is idempotent. It never changes the state of the web service but can be compared to a read-only getter method returning an array of items. The *sendPurchaseOrder* operation places an order for the purchase items and returns an invoice. The input of this operation is the information about the customer (customer name, address, etc.) and the order itself (the items including id, name, and price, among others). The state of the web service is changed when a new order is placed.

In the purchase order scenario regarded from the gray box view, the internal process logic becomes visible. Figure 4.4 shows the process that is started when the *sendPurchaseOrder* operation is called. This operation is bound to the first *receiveTask* and the last *replyTask*. Receive tasks usually have an implication on the WSDL interface of the service. For example, the *receiveInvoice* task is bound to the *sendInvoice* operation which allows sending the invoice information from an external partner service to the process.

In the white box view, the WSDL file and the code for the implementation is available including all internal classes and methods being used. It is assumed that the purchase order service has been completely implemented in Java. This is necessary in order to use NFComp's reverse-engineering code-to-model transformation. For a better understandability, the class design of the service is assumed as follows: There is one service class per port type implementing one method for each operation. For each partner service there is a stub class implementing the inter-

face of the partner service. This stub is responsible for transforming the Java method calls into SOAP invocations to a particular partner web service. The stub is called from within the service classes. To enable modeling of the composition of NFCs with help of NFComp, a graphical model is required. This graphical model must be generated out of the Java code in this case (although it also possible to model this manually). The functional model must be available before the action to application mapping phase.

Figure 4.5 shows the BPMN2 process generated out of the Java classes depicted in Figure 4.3. There are several pools, each for a particular method that has been selected for behavior generation. The first pool represents the *sendPurchaseOrder* operation of the *PurchaseOrderPT* class and shows the invocation of the different stubs. The process is sequential because there is no support for parallelism in the generator yet. The second operation of the *PurchaseOrderPT* class, *getItemsByCategory* is not shown in the process. It invokes the *ItemStore* class, which provides access to the item database. The second pool represents the *getAvailableItemsByCategory* method of the *ItemStore*. Firstly, the validity of the category is checked. This is accomplished by the *isCategoryAvailable* method represented by the third pool. If the category exists, a named query will be executed in order to load all items from the corresponding category from the database. If the category does not exist, it will be created.

4.4.1 Requirements Specification

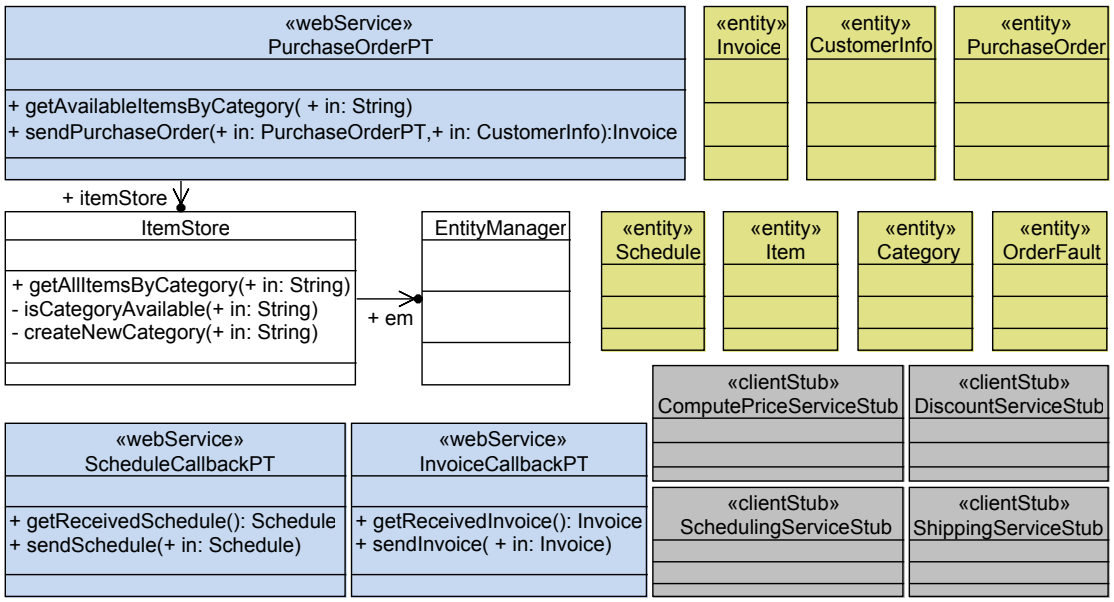

Figure 4.3: Classes Implementing the Purchase Order Process

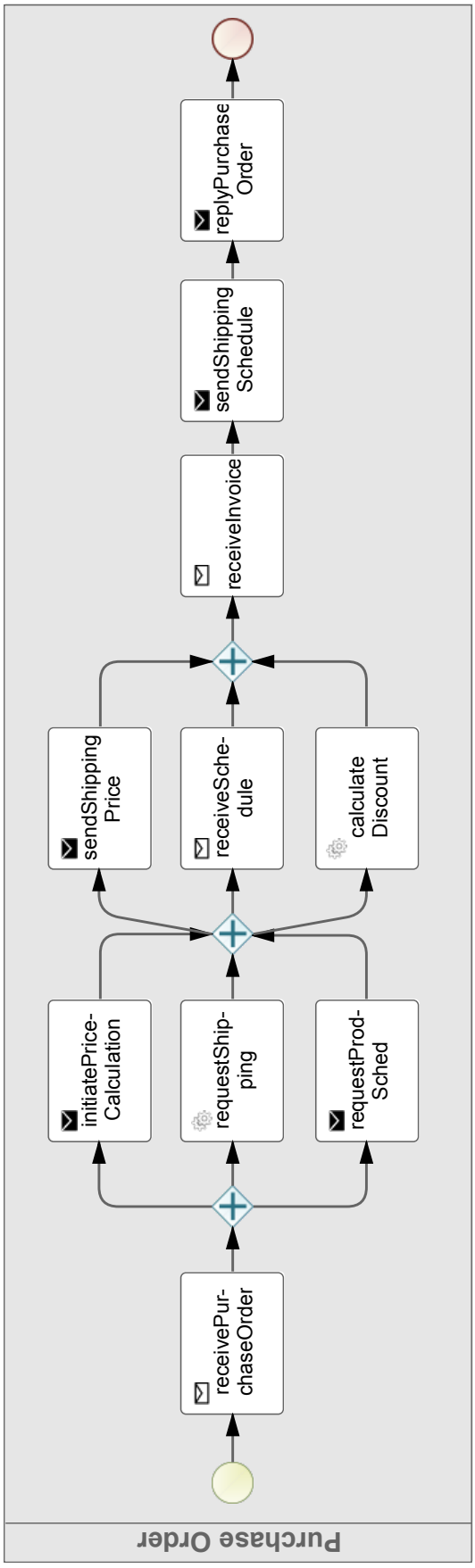

Figure 4.4: Process Logic of the Purchase Order Service

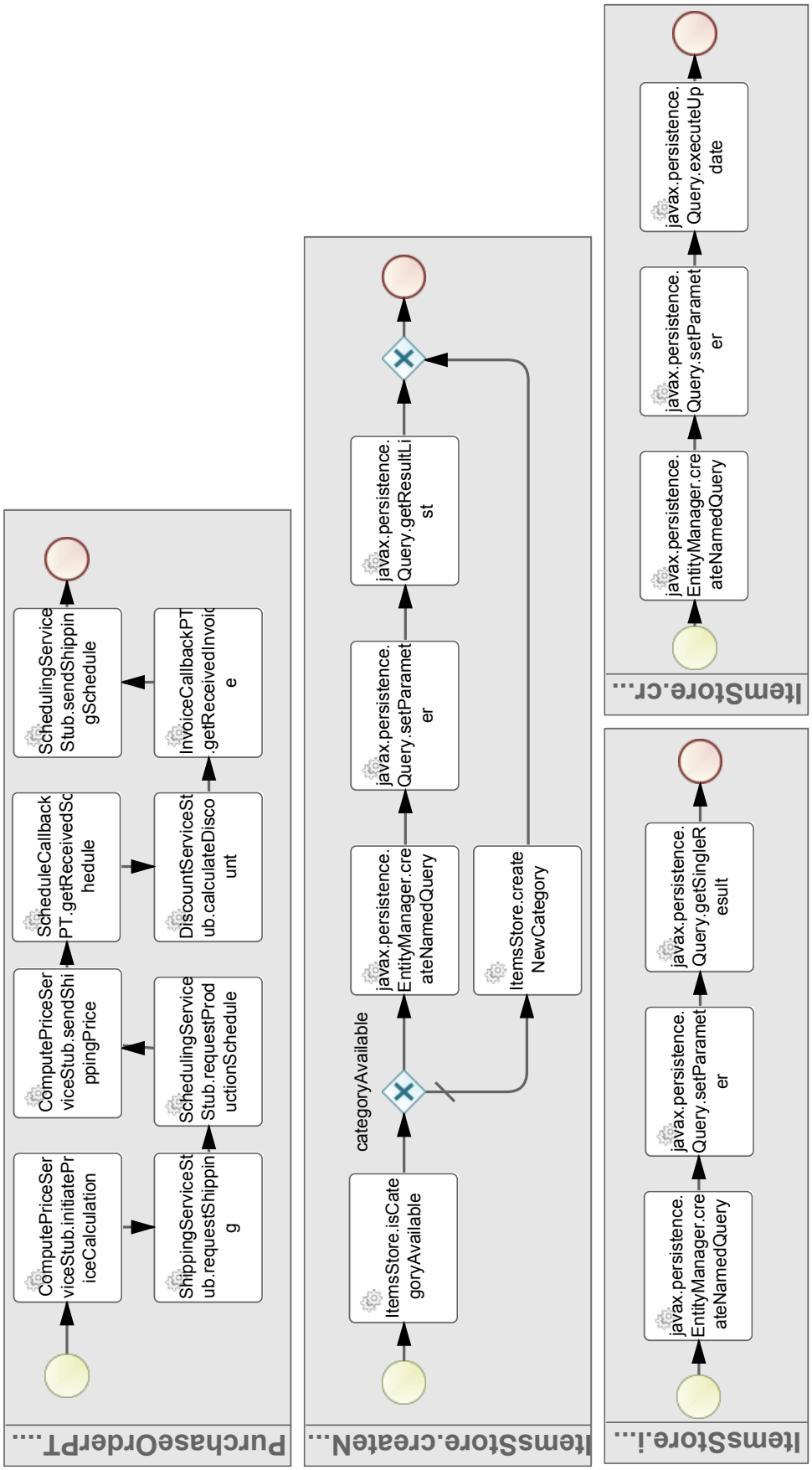

Figure 4.5: Process Logic Reverse-Engineered From Purchase Order Java Implementation

4.4.1.1 Case Study

The non-functional requirements to be addressed for the scenario are presented in the following. The requirements in the black box view are listed here:

- The customer and invoice data should be confidential and not readable by unauthorized parties (Security).
- The order and invoice data should not be modifiable by, for example, any kinds of man-in-the-middle-attacks (Security).
- The purchase order operation can only be called by authenticated consumers (Security).
- The purchase order operation should not be invoked multiple times due to possible message duplication (Reliable Messaging).
- There should be a purchase history for each customer (Logging).
- The *getAvailableItemsByCategory* operation is frequently used by many different categories of customers and must perform well (Performance).

Switching to the gray box view reveals additional requirements (which are also important from the white box view):

- The *requestProductionSchedule* message should be guaranteed to arrive at the receiver before the *shippingPrice* message arrives (Reliable Messaging).
- The discount calculation is provided by a possibly slow partner service. Execution time should be measured for this service invocation (Monitoring).
- The execution time of the purchase order process should be measured (Monitoring).
- All invocations of partner services should be logged (Logging).

In the white box view, all code artifacts are visible and thus also components which are hidden behind the service interface. Figure 4.3 shows that there are classes which are implementing the service interface (annotated with the «webService» stereotype), as well as classes that are involved in this implementation by realizing a particular unit of work. One example is the *ItemStore* class which is responsible for database access. This data access is not read-only. The *DataStore* class will insert a new category, when it does not exist. This operation should be executed in an atomic transaction.

- All operations with write access to the database should be executed in a transaction (Transactions).
- The execution time of database operations with read access should be monitored (Monitoring).

NFComp In NFComp, the requirements are explicitly defined and thus part of the specification model, as can be seen in Figure 4.6. The requirements engineer has modeled concern boxes for security, logging, monitoring, performance and reliable messaging. This decision helps to separate the concerns in general so that respective non-functional domain experts can be found and the action model can be separated accordingly. For example, there could be different action model files for each concern. The concern is also an important grouping element for the non-functional requirements. Each non-functional attribute has been assigned to a particular concern and is contained in the respective box. For each attribute an additional description can be added to make the attribute more understandable. The attributes are later imported into the action definition model in order to be able to associate them with the actions that realize them.

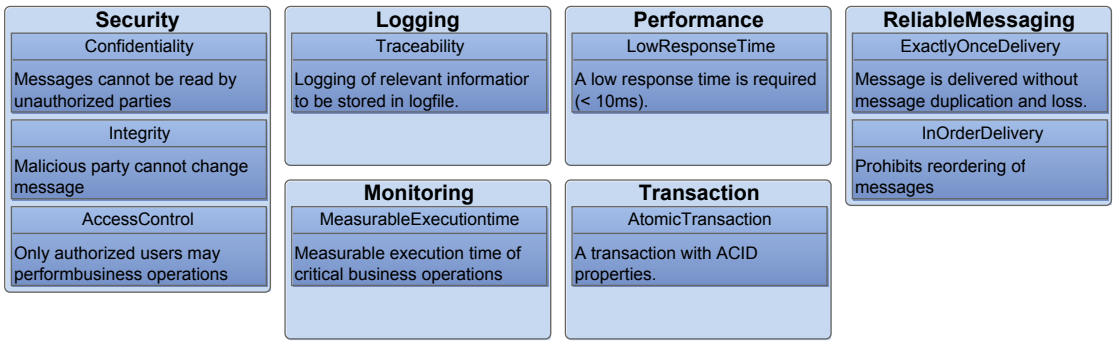

Figure 4.6: NFComp Requirements Model for Purchase Order

WS-Policy/SCA When using pure industrial standards, the WS-Policy [95] framework is the web services standard for specifying requirements, capabilities and other properties for web services. However, there is only one concept — the assertion — that must be used for all types of specification items. Thus, there is no clear separation of requirements and actions. WS-Policy comprises a family of domain-specific specifications such as WS-SecurityPolicy, and WS-ReliableMessagingPolicy, among others. In these specifications, actions such as encryption, message signing etc. have been defined which correspond more to the concept of an action rather than a requirement. However, requirements and capabilities in the WS-Policy context rather mean that a web service has the requirement to use encryption in order to communicate with it or has the capability to send encrypted messages. Consequently, WS-Policy is more appropriate for action definition. This drawback, though, can be addressed by using the SCA [5] standard. SCA makes use of WS-Policy but extends policies by *intents* representing abstract requirements such as *authentication*. These *intends* are later bound to concrete actions realizing them. Listing 4.1 shows a subset of the purchase requirements specified as intends and mapped to actions using *intentMaps* and policy attachments.

```
1  ...
2  <intent name="sca:Confidentiality" constrains="sca:binding">
3    <description>Message cannot be read by unauthorized parties</description>
4  </intent>
```

```
5   <intent name="sca:Security" constrains="sca:binding"
6     requires=" sca:Confidentiality sca:Integrity sca:AccessControl ">
7   <description>Security Requirement</description>
8   </intent>
9   <intentMap provides=" sca:confidentiality " default=" transport ">
10    <qualifier name="transport">
11      <wsp:PolicyAttachment>
12        <wsp:AppliesTo>
13          <wsa:EndpointReference>
14            <wsa:Address>http: // purchaseorder .com/purchaseorder</wsa:Address>
15            <wsa:PortType>purchaseOrderPT</wsa:PortType>
16            <wsa:ServiceName>PurchaseOrderService</wsa:ServiceName>
17          </wsa:EndpointReference>
18        </wsp:AppliesTo>
19        <wsp:PolicyReference
20          URI="http: // purchaseorder .com/purchaseorder/ policies #SecPolicy" />
21      </wsp:PolicyAttachment>
22    </ qualifier >
23  </intentMap>
24  <service>
25    <binding. binding-type requires=" sca:confidentiality ">
26  </service>
```

Listing 4.1: Requirements Specification With Intents in SCA

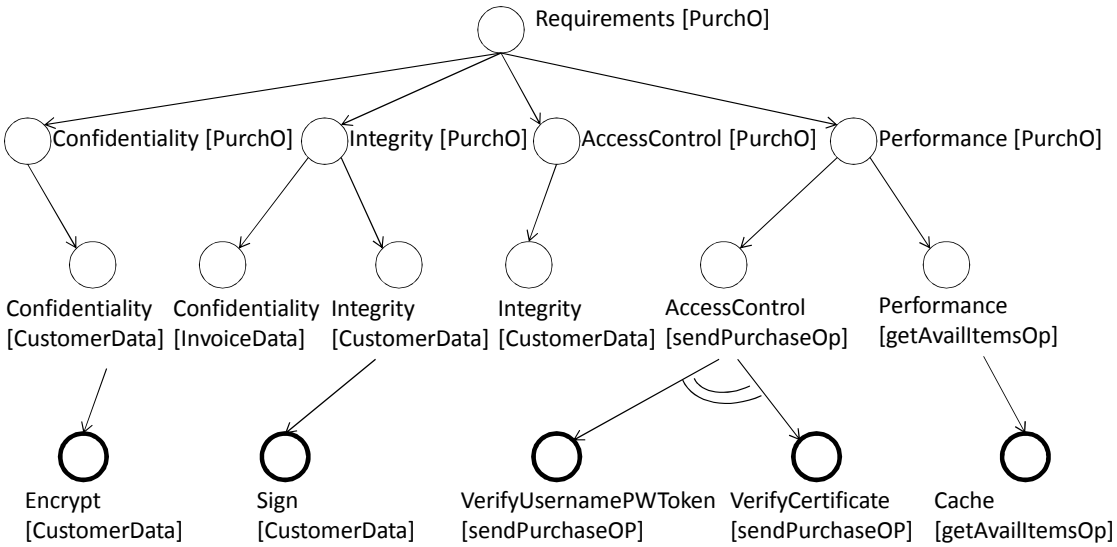

Figure 4.7: Requirements Modeled in the NFR Framework

NFR Framework There are many approaches that allow for the specification of non-functional requirements. A prominent one is the NFR Framework [18]. It provides its own model in terms of softgoals comparable to non-functional requirements and satisficing goals comparable to actions. Figure 4.7 shows a subset of the requirements for the purchase order service defined in the NFR Framework. It forms a tree of circles representing softgoals. In case of the purchase order

scenario, there could be, for example, one root node *Requirements* for purchase order which is then further refined. In the next level there are the particular requirements such as confidentiality, integrity, access control and performance. The next refinement level shows in more detail which soft goal pertains to which resource; e.g., confidentiality and integrity are required for customer and invoice data, and access control is required for the *sendPurchaseOrder* operation. In the next level, satisficing goals (represented by bold circles) are shown describing how a particular softgoal can be satisfied. For access control there are two possible satisficing goals the verification of the consumer's username and password or the verification of the consumer's certificate. The two circles between the lines define that the username and password OR the certificate can be used. A single line (AND) would define that both satisficing goals are needed to satisfy a softgoal.

Sec-MoSC In the Sec-MoSC approach [88] security-related non-functional requirements are modeled in terms of *NF-Attributes* which are either composite (Security) or primitive (Integrity, Confidentiality). Each attribute is associated with one or many (by duplicating the attribute) BPMN process tasks in order to express the requirements for that task. An association is called *NF-Bind*. An *NF-Attribute* is additionally associated with a set of *NF-Actions* expressing how the attributes are to be realized. Furthermore, there is the concept of an NF-Statement modeling constraints on an attribute. For security, multiple levels (*high*, *medium*, *low*) have been defined which can be annotated with the cloud symbol which represents an *NF-Attribute*. Figure 4.8 depicts some of the requirements for purchase order modeled as *NF-Attributes* and bound to the respective process tasks.

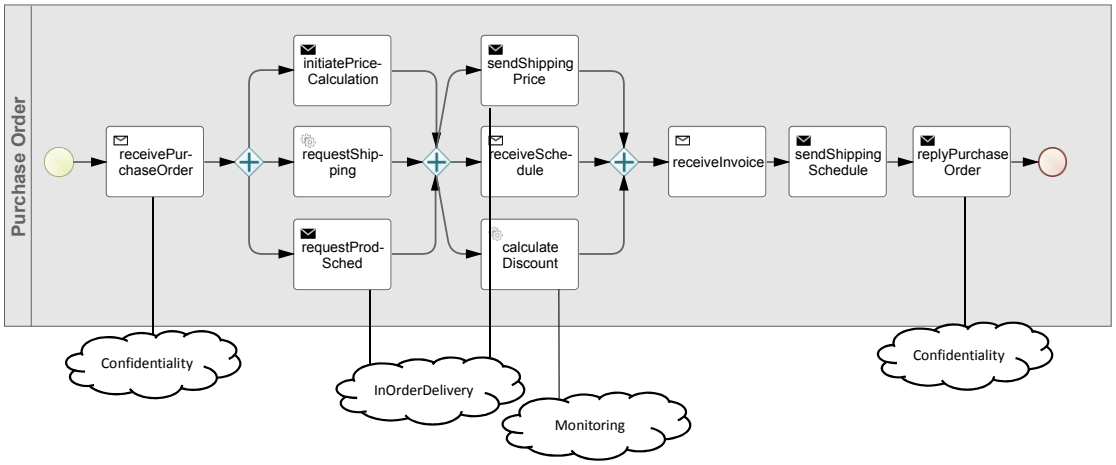

Figure 4.8: Requirements Modeled in Sec-MoSC [88]

ProcessNFL ProcessNFL [73] can also be used to define requirements in form of attributes and actions. Listing 4.2 shows one possible example for the purchase order process. Firstly, there are different attributes. These attributes can be composite and further decomposed;

for example, security (Line 8-12) can be decomposed into the primitive attributes *integrity*, *confidentiality* and *accessControl* (*primitives* keyword, Line 10). The *contribution* keyword (Line 11) defines how many of the primitives must be satisfied in order to satisfy the composite attribute. An action can be related to an attribute either via the *implemented* keyword (Line 23) or the *affected* keyword (Line 29) depending on the impact of an action. In case of *affected* the attribute is not completely satisfied by performing the action but there is either a positive or negative effect on the attribute (between -3 and +3, for example). Finally, properties (Line 34-42) define constraints on the attributes; e.g., performance and security must be at least medium and priority of security is higher than that of performance.

```
1  attribute performance extends NFR;
2    {
3      primitives none;
4      contribution none;
5    }
6  attribute security extends NFR;
7    {
8      primitives integrity , confidentiality , accessControl ;
9      contribution all ;
10   }
11 attribute confidentiality extends security ;
12   {
13     primitives none;
14     contribution none;
15   }
16 action encrypt ;
17   {
18     affected none;
19     implemented confidentiality ;
20     effect none;
21   }
22 action readFromCache;
23   {
24       affected performance;
25       implemented none;
26       effect performance [+2];
27   }
28 property mediumPerformanceAndSecurity;
29   {
30     constraints
31       performance [ medium ];
32       security [ medium ];
33     priorities
34       performance [ low ];
35       security [ high ];
36 }
```

Listing 4.2: Purchase Order Requirements Described in ProcessNFL [73]

AO4BPEL In AO4BPEL [14] the deployment descriptor can be used to specify the non-functional requirements for composite web services. The deployment descriptor contains

selectors to associate requirements with subjects such as WS-BPEL activities and *services* for the different middleware services supported by the framework. There are four middleware services; one for security, one for reliability, one for logging and one for transactions. These services support a set of requirements from different predefined classes such as *confidentiality* or *semantics* (shortcut for delivery semantics), and each requirement has a *type*. This type can be selected from a predefined set of types for each class. Examples for confidentiality types are encryption and decryption, for example. Listing 4.3 shows what the deployment descriptor would look like for the purchase order process.

```
<bpel-dd>
  ...
  <services>
    <service name="reliablemessaging">
      <requirements>
        <requirement name="req0" class="semantics" type="inOrder"/>
      </requirements>
    </service>
    <service name="security">
      <requirements>
        <requirement name="req1" class=" confidentiality " type="encryption">
          <parameters>
            <parameter name="symmetricEncAlgorithm">xmlenc#tripledes-cbc</parameter>
            <parameter name="keyEnc">http://www.w3.org/2001/04/xmlenc#rsa-1_5</parameter>
            <parameter name="transportKeyId">16c73ab6-b992-458f-abe5-2f875f77882e</parameter>
            <parameter name="keyIdentifierType">-1</parameter>
          </parameters>
        </requirement>
        <requirement name="req2" class=" confidentiality " type="decrypt" />
        <requirement name="req3" class=" integrity " type="sign" />
        <requirement name="req4" class=" integrity " type="checkSignature"/>
        <requirement name="req5" class=" authentication " type="usernametoken" />
      </requirements>
    </service>
    <service name="logging">
      <requirements>
        <requirement name="req6" class="message" type="request"/>
        <requirement name="req7" class="message"type="response"/>
      </requirements>
    </service>
  </services>
</bpel-dd>
```

Listing 4.3: AO4BPEL Deployment Descriptor

4.4.1.2 Feature Comparison

Criteria The analysis of NFComp and related work revealed a set of features or criteria which can be used to conduct an objective comparison. The distinction of requirements and actions is an important feature because it allows the determination of requirements independently of the underlying platform. The separation of requirements and actions into different model files helps

to reuse requirement specifications throughout different scenarios/projects. Requirements can be specified on different abstraction levels. Thus, it should be possible to decompose an abstract requirement into more concrete requirements. In terms of feature points this means that arbitrary depth of decomposition is rated with 1 and one level of decomposition is rated with 0.5 points. Actions and requirements are associated with each other. The multiplicity of this association has an impact on the reusability of requirements and the expressiveness. If, for example, an n-m relationship is possible, a requirement can be associated with several actions and vice versa. A n-m relationship is rated with 1 feature point, whereas 1-n is rated with 0.5. The kind of metamodel is also an important factor. If the metamodel adheres to standards, it is easier to be integrated with other metamodels, and also the chances that a suitable toolset for its creation is available is much higher. The metamodel defines the abstract syntax for the model and hence helps to validate the model with respect to its syntax. The use of standard metamodel (XML Schema, Ecore, UML) is rated with 1 feature point. In the approaches that have been analyzed, different types of requirement descriptions have been used, either an informal description or a leveled (for instance, low, medium, high) one. The description types have not been rated, because a leveled description is not necessarily better than a informal one (and vice versa). Another feature to be analyzed is whether the approach supports a mapping from a requirement to the respective target subject. A direct mapping is rated with 1 feature point, whereas an indirect one is rated with 0.5. Particular approaches already provide a predefined set of requirements that can be used out-of-the box, which is generally desirable. However, it is also important to support custom requirements that are not part of the predefined set of requirements.

NFComp NFComp separates requirements from actions; both are defined in separate models whereby the action model imports one or more requirement models. A requirement can be decomposed, but only one level of decomposition is supported. This is covered by the concept of a non-functional concern (such as security or reliability), which is decomposed into non-functional attributes. The association between actions and requirements is 1-n, an action can be associated with one attribute and one attribute with several actions. In addition, also the type of the association (satisfy, contribute positively, contribute negatively, deny) can be specified. The metamodel is a formalized Ecore model. The requirements specification is informal. The mapping of requirements to their target is indirect. Not the non-functional attribute, but the concrete action is associated with the target (web service, process tasks and others). The attribute is associated with the action, so that the relationship between target and requirement can be derived. NFComp does not define a set of predefined requirements but allows the specification of any kind of requirement. To support the satisfaction of a new requirement, a new action must be defined.

WS-Policy/SCA WS-Policy [95] does not distinguish requirements from actions. In the majority of domain-specific WS-Policy specifications, concrete actions are defined in contrast to requirements. This deficiency can be compensated for by using SCA. In SCA there is a dis-

crimination of requirements and actions, and both are separated into different models. Actions can be specified in separate policy documents which can be attached to the *intends* using WS-PolicyAttachment. This and the *intentMap* allow an n-m mapping between actions and requirements. The mapping is not typed; there is only a general association between intents and policies. Another drawback, inherited from WS-Policy, is that the XML Schema used to define the syntax of the policy document cannot be used to validate domain-specific policies (cf. Heinzl et al. [41]). Intents and thus requirements can be directly mapped to their target, e.g., a service. A predefined set of security, reliability and transactions *intents* is supported. The problem with SCA, however, is that it has not been widely adopted yet and its usage may introduce unnecessary complexity.

NFR Framework The NFR framework [18] supports only a few concepts but distinguishes softgoals (requirements) from satisficing goals (actions). Only a single model is used for the specification of soft- and satisficing goals. Requirements can be decomposed arbitrarily. The association between soft and satisficing goals is n-m and it is allowed to use AND/OR expression to describe the relationship. There are also different types of associations: A softgoal may be satisficed, denied, satisficable or deniable by a satisficing goal. No official, formalized metamodel for NFR is available; however Supakkul and Chung [89] proposed a UML profile for representing the NFR Framework in UML Use Case and Class Diagrams. The requirements are described informally and are mapped to the target resource directly. In theory, the NFR framework is able to replace the simplified NFComp requirements model. The prerequisite for this is that the NFR framework has been formalized into a machine processable format such as XML. However, a much easier integration would be possible if there were a metamodel for the framework; e.g., for NFR Framework. If this metamodel had additionally been based on Ecore, the NFComp could directly link to elements in this metamodel.

Sec-MoSC In Sec-MoSC [88], there is a clear separation between requirements (*NF-Attributes*) and actions (*NF-Actions*). They are specified in one and the same model and can be decomposed (the authors distinguish composite and atomic attributes). The association between requirements is 1-n; one requirement can be associated with multiple actions. Only a general association between requirements and actions is possible; no types are used to further describe the nature of the association. *NF-Statements* allow the constraint of the attributes by certain levels. A metamodel defined in Ecore is available, and requirements are directly associated with process tasks. A predefined set of nine requirements (for example, *Confidentiality*, *Data Retention*, *Access Control*, *Authentication*, etc.) for the security domain is offered which are also supported at runtime. If a new requirement needs to be introduced, new actions may also need to be introduced. When a new action is to be supported, the generator needs to be extended to generate appropriate WS-Policy configuration files. Thus, Sec-MoSC is not suitable for all kinds of actions but focuses more on its predefined security actions.

ProcessNFL ProcessNFL [73] separates requirements from actions, but both are defined in the same model. Attributes can be further decomposed using the *extends* keyword. An action can be associated with multiple attributes, and an attribute can be used in different actions (n-m relationship). Different association types are supported, such as *affected* and *implemented*. No formalized metamodel is supported since a custom textual notation has been chosen for this approach. The requirements description is informal. For each association between attribute and action, the level of effect can be specified. There are no subjects an attribute or action can be mapped to. ProcessNFL does not come up with a predefined set of requirements but allows the modeling of any kind of requirement without restrictions because there is no runtime enforcement support.

AO4BPEL AO4BPEL [14] has a separation of requirements and actions but this is not very obvious. The actions are hidden in the type attribute of a requirement definition in the deployment descriptor. They are tightly connected to requirements. Only one level of requirements decomposition is supported: Firstly, a requirement is a descendant of the *service* element, which is comparable to NFComp's non-functional concern concept. The requirement is of a certain class (*class* attribute) and can further be detailed with the *type* attribute corresponding more to the non-functional action concept. There is a 1-n association between actions and requirements: A requirement can be of different type. Association types are not supported. The requirements metamodel is based on XML Schema; however, the different classes per requirements and types per classes cannot be validated against this schema. This could lead to conflicts when generating aspects out of the deployment descriptor. There is an informal requirements definition. Requirements can be directly associated with WS-BPEL activities via *selectors*. There is a predefined, default set of requirements that can be used to generate aspects enforcing the requirements at runtime. However, to implement a new requirement, the generator must be aware of the new requirement-service-class-type combination and generate appropriate aspects. This means, that whenever there is a new requirement to be supported, new aspect generation code (defined in XSLT) must be written.

Summary In summary, the advantage of NFComp over WS-Policy is that no technical knowledge is required during the requirements phase. This helps on one hand to involve people like dedicated requirements engineers, who are not technical experts, in the process, and on the other hand to improve flexibility by maintaining independence from the underlying platform/technology as long as possible. NFAs can be defined by respective non-functional domain experts later, depending for example on the concrete execution environment or platform, use case or other factors. Compared to GORE frameworks, NFComp is holistic and covers different aspects of non-functional concerns composition for web services, not only requirements and actions. This allows the tight integration of the different phases and their models, such as requirements specification, action definition, action composition and mapping with each other. However, GORE approaches such as the NFR framework are more powerful because they support AND/OR ex-

pressions between associations of different types. In fact, NFComp supports different association types, though there is no m-to-n relationship between attributes and associations; i.e., an action cannot be associated with multiple attributes, which is possible for example in NFR framework. Sec-MoSC proposes a slightly different approach: It associates process tasks with non-functional attributes. NFComp associates actions with process tasks and requirements with actions. However, with respect to the requirements phase, both approaches are quite similar.

Table 4.7 shows an overview of the features discussed. Features in the table which have not been rated are displayed with an italic font type. NFComp is not the approach with the most feature points. The strongest approach in this category is SCA/WS-Policy with 7.5 of 9 feature points. It supports all features except requirement metamodel and different association types.

Feature/Crit.	**NFComp**	**NFR**	**Sec-MoSC**	**SCA**	**ProcNFL**	**AO4BPEL**
Distinction of reqs & actions	Yes	Yes	Yes	Yes	Yes	Yes
Model separ. of reqs & act	Yes	No	No	Yes	No	No
Decomposition of reqs	1 Lvl	Yes	Yes	Yes	Yes	1 Lvl
Associations support	1-n	n-m (expr)	1-n	n-m	n-m	1-n
Different assoc types	Yes	Yes	No	No	Yes	No
Requirement metamodel	Ecore	UML Profile	Ecore	(Schema)	No	(Schema)
Requirement description	Informal	Informal	Leveled	Informal	Leveled	Informal
Mapping reqs to target	Indirectly	Yes	Yes	Yes	No	Yes
Predefined set of reqs	No	No	Yes	Yes	No	Yes
Generic reqs	Yes	Yes	No	Yes	Yes	No
Feature Points	6.5	7	5.5	7.5	5	4.5

Table 4.7: Feature Comparison in the Requirements Phase

4.4.2 Action Definition

4.4.2.1 Case Study

NFComp In NFComp, the action definition task is performed by a distinct role, the non-functional domain expert. For each non-functional domain — such as security, performance, reliability, transactions and so on — a responsible expert is assumed. In NFComp arbitrary non-functional domains and actions can be defined.

The security expert knows how to achieve confidentiality and integrity for web services.

She defines two actions, one for encrypting messages that are sent by the service (satisfying the confidentiality attribute), and one for signing messages (satisfying the integrity attribute). Moreover, she adds decryption and verification of signatures as actions for processing incoming messages which are signed or encrypted. The resulting model is shown in Figure 4.9. Additionally, she defines the properties of these actions. Table 4.8 shows the properties of the *Encrypt* action, exemplarily. The *Encrypt* action satisfies the attribute confidentiality. It can only be applied to outgoing messages. The impact on the message is that it modifies the message, more specifically its body.

Property Name	**Property Value**
Name	Encrypt
Direction	Out
Attribute	Confidentiality (Satisfies)
Impact	Modify
Target	Body

Table 4.8: Properties of the Encrypt Action

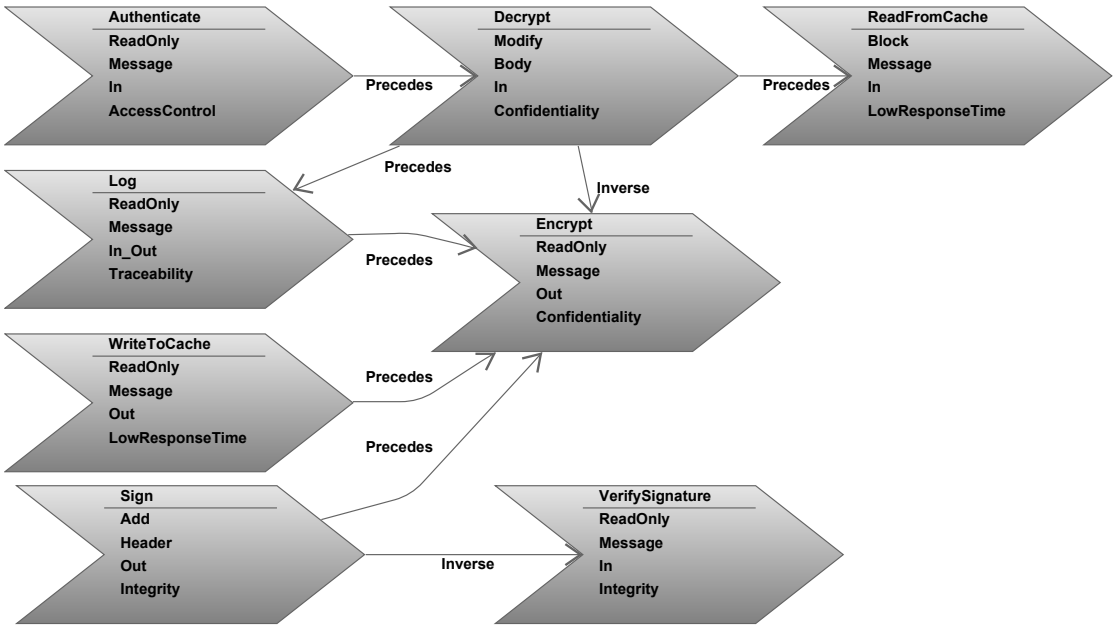

Figure 4.9: A Subset of Actions for the Purchase Order Service

The security expert decides to use plain (unencrypted) signatures and not to encrypt the header of the message but only its body. Moreover, the signature should include the plain body and not the encrypted one. Hence, she defines the *precedes* interdependency as follows: *precedes(Sign, Encrypt)*. Furthermore, she defines: *inverse(Encrypt, Decrypt)* and *inverse(Sign, VerifySignature)*. The logging expert defines an action *Log*, and the performance expert defines two actions, one for writing frequent messages to a cache and one for retrieving the messages from the cache using the parameters of the message as key. Due to the declaration

of the *inverse* interdependencies, *precedes(Decrypt, VerifySignature)* can be implicitly inferred from the model and is not required to be modeled explicitly.

After defining the actions from different non-functional domains, the experts (in a collaborative manner) should determine interdependencies across non-functional domains. They can use the impact properties to systematically determine interdependencies; e.g., the *Encrypt* action will change the body into an unreadable (by unauthorized parties) string having an impact on all actions that need to read the body; for example, *ReadFromCache* requires the capability to read the parameters. By this methodology the following cross-domain interdependencies can be found: *precedes(Decrypt, ReadFromCache)*, *precedes(Authenticate, Decrypt)*, *precedes(WriteToCache, Encrypt)*, *precedes(Log, Encrypt)*, *precedes(Decrypt, Log)*. The resulting action definition model is depicted in Figure 4.9.

WS-Policy WS-Policy [95] is the web services standard for the specification of non-functional capabilities or requirements in the form of actions to be performed. For example, the WS-SecurityPolicy defines a set of security actions that can be used to secure messages at the SOAP level. WS-Policy offers a set of standardized actions for addressing (WS-Addressing [9]), reliable messaging (WS-ReliableMessagingPolicy [27]), security (WS-SecurityPolicy [55]), and atomic transactions (WS-AtomicTransactions [56]) among others. These policies are also referred to as domain-specific policies. The WS-Policy specification, hence defines a standardized model for all domain-specific specifications. Listing 4.4 (wssp = WS-SecurityPolicy, wsrmp = WS-ReliableMessagingPolicy) shows a simplified example (some elements have been omitted) of how a policy document could look for the purchase order service. There are four top-level assertions that have been defined. The first assertion *RMAssertion* (Line 4-10) is from the WS-ReliableMessaging specification and prescribes that messages must be delivered with the delivery assurance *AtMostOnce*, i.e., declining duplicate messages. The second assertion *SupportingTokens* (Line 11-15) is a container for required tokens such as a *UsernameToken* which can be used for authentication. The third assertion *SignedElements* (Line 16-18) defines the elements of the message to be signed, and finally the fourth assertion *EncryptedElements* (Line 19-21) defines which elements of the message shall be encrypted. The *ramp* namespace elements (Line 22-39) represent configuration options specific to the security module being used, Apache Rampart[8] in this case.

```
1  <wsp:Policy wsu:Id="POPolicy">
2   <wsp:ExactlyOne>
3    <wsp:All>
4     <wsrmp:RMAssertion>
5      <wsp:Policy>
6       <wsrmp:DeliveryAssurance>
7        <wsp:Policy><wsrmp:AtMostOnce/></wsp:Policy>
8       </wsrmp:DeliveryAssurance>
9      </wsp:Policy>
10    </wsrmp:RMAssertion>
```

[8] http://axis.apache.org/axis2/java/rampart/

```
11      <wssp:SupportingTokens>
12        <wsp:Policy>
13          <wssp:UsernameToken wssp:IncludeToken=".../IncludeToken/Once" />
14        </wsp:Policy>
15      </wssp:SupportingTokens>
16      <wssp:SignedElements>
17        <wssp:XPath>/S:Envelope/S:Body</wssp:XPath>
18    </wssp:SignedElements>
19    <wssp:EncryptedElements>
20        <wsssp:XPath>/S:Envelope/S:Body</wssp:XPath>
21      </wssp:EncryptedElements>
22      <ramp:RampartConfig xmlns:ramp="http://ws.apache.org/rampart/policy">
23        <ramp:encryptionUser>serverkey</ramp:encryptionUser>
24        <ramp:encryptionCypto>
25          <ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
26            <ramp:property
27              name="org.apache.ws.security.crypto.merlin.keystore.type">
28              JKS
29            </ramp:property>
30            <ramp:property name="org.apache.ws.security.crypto.merlin.file">
31              pathToFile
32            </ramp:property>
33            <ramp:property
34              name="org.apache.ws.security.crypto.merlin.keystore.password">
35              password
36            </ramp:property>
37          </ramp:crypto>
38        </ramp:encryptionCypto>
39      </ramp:RampartConfig>
40    </wsp:All>
41  </wsp:ExactlyOne>
42 </wsp:Policy>
```

Listing 4.4: WS-Policy With Assertions From Different Domains

In general, there are textual and model-driven approaches to specify non-functional actions. ProcessNFL [73] Rosa et al. is a representative for a textual language to specify non-functional attributes, actions and properties. This language has already been investigated sufficiently in the requirements specification phase.

Ortiz and Hernandez In model-driven approaches, non-functional actions are often modeled as properties that are attached to the model used for code generation, such as a UML Class Diagram. Based on these non-functional properties, additional code is generated for enforcing the modeled properties. For example, Ortiz and Hernandez [68] specify an abstract UML Stereotype called *Extra-Functional Property* representing a non-functional action. It defines different properties: an *actionType* which can be one of before, after, instead or none; the *optional* attribute (true or false); the *priority* attribute defining the execution order at runtime, the *policyid* to refer to a WS-Policy and the *ack* attribute indicating if full code or only skeletons can be generated.

The abstract stereotype is extended by other concrete stereotypes representing specific extra-functional properties. These concrete stereotypes inherit the properties of *Extra-Functional Property*, but can define their own specific properties, for example the name of the *logFile* in case of logging. These concrete stereotypes can be applied to operations or interfaces of UML classes representing web services. Figure 4.10 shows how the UML Profile can be extended by own use-case-specific stereotypes. The resulting profile(s) can be applied to the class diagram representing the purchase order service. This allows applying the stereotypes to interfaces and operations and definition of the values for the properties of the respective stereotype. On one hand, it is possible to model any kind of action; on the other hand, there is only a small subset of extra-functional properties supported for which to generate executable code. For properties not directly supported, the generated aspects are just skeletons and must be extended manually.

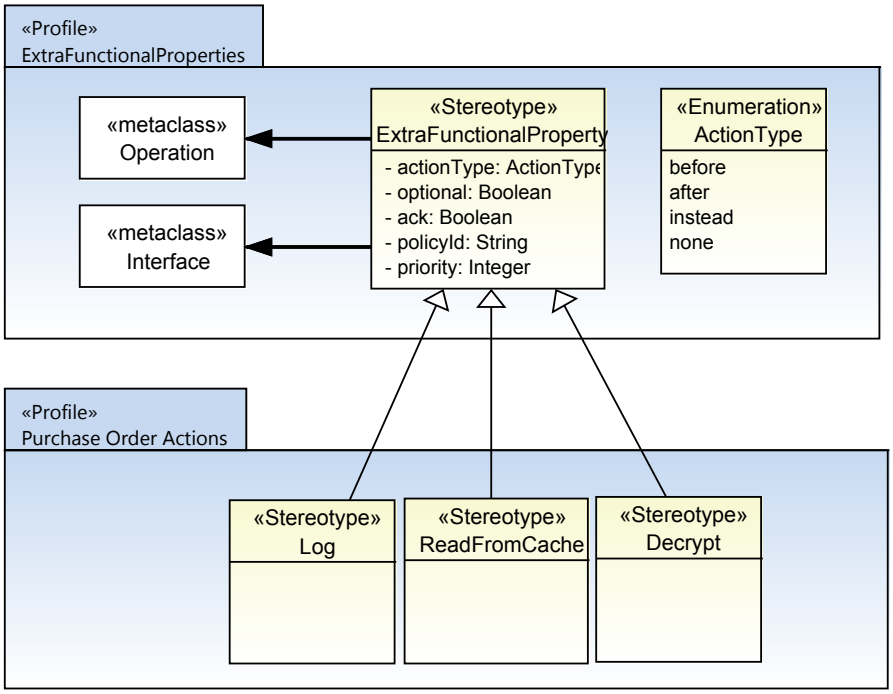

Figure 4.10: Extra-Functional Property Stereotypes for Purchase Order Service in the Approach of Ortiz and Hernandez

Sec-MoSC In Sec-MoSC [88], NF-Actions can be associated with NF-Attributes via NF-Bind associations. Each NF-Action may have a set of properties for the configuration of its realization. Properties are tuples <*name, value*> such as <*encryption type, symmetric*> for example. For purchase order, the supported actions can be applied: *UseCryptography*, *UseDigitalSignatures* and *Log*. For supporting the other actions, new handlers must be written and the generator must be extended.

AO4BPEL As already mentioned in the last section, actions in the AO4BPEL [14] deployment descriptor are tightly bound to requirements. There are different actions depending on the requirements, e.g., encryption, decryption and so on. Actions can be parametrized to configure them accordingly. The AO4BPEL deployment descriptor for the purchase order process is shown in Listing 4.3. The actions for reliable messaging and security have been used, and configuration parameters have been set accordingly. For the other actions such as caching, there is no out-of-the-box support in AO4BPEL. For this purpose, new middleware services must be developed and the deployment descriptor must be extended by new *requirements*, *classes* and *types*. Furthermore, the aspect generator must be extended (see discussion in requirements phase).

4.4.2.2 Feature Comparison

Criteria The application of NFComp and related work to the purchase order scenario revealed a set of desirable features that can be used for a more detailed comparison. Firstly, the granularity of actions is an important factor. An action can be fined-grained so that there is a distinction between actions that are applied to incoming and those applied to outgoing messages or coarse-grained so that there is no such distinction. The more fine-grained actions are, the more powerful and flexible the composition. If, for example, *Encrypt* and *Decrypt* are modeled separately, they can be composed with other actions independently from each other. If there is only one action such as *UseCryptography*, this will not be possible.

A theoretical example would be to have three actions, A, B and C, for incoming messages; two, D and E, for outgoing messages; and one, F, for both. If actions B and E were two fine-grained actions inverse to each other, they could be composed independently, for example, in the following order: A->F->B->C and D->E->F. Action F can be positioned flexibly before or after actions B and E, which would not be possible if considered only as one coarse-grained action BE.

This is true especially for those approaches that define general orderings of actions not specific to the target or direction (in NFComp, one could have defined a separate order for each direction, each containing the action BE, whereas in other approaches this would not be possible). If the action is specified in a fine-grained fashion, the application of an action can also be restricted to a certain direction, e.g., only to incoming or outgoing messages. The description of the impact of an action helps to compose it with other actions. If this is done in a formal way, it will also be possible to validate action compositions based on the impact description. An action could be described in a very fine-grained way, though the behavior of one and the same action could be configurable. Otherwise, for each configuration a distinct action must be modeled. For example for an *Encrypt* action, the encryption algorithm and key strength should be configurable. If there is no support to specify configuration parameters, there would be different types of *Encrypt* actions such as *EncryptWithAES256*, *EncryptWithRSA* and so on. Although this may be a viable modeling solution, it should generally be possible to summarize

these actions as one *Encrypt* action, because regardless of which key or algorithm is used, the properties of these actions are equal. Like for requirements specification, the metamodel plays an important role for action definition and is rated as in the previous subsection. Another feature for action definition is the specification of interdependencies between actions. This allows the validation of the modeled composition against constraints imposed by the interdependencies.

NFComp In NFComp, the action granularity is fine. There is a difference between actions that can be applied before or after a functional subject, e.g., *Encrypt* and *Decrypt*. The application of an action can be constrained by the functional point it is applicable to, e.g., only to incoming or outgoing or fault messages. Unlike WS-Policy, NFComp does not provide a predefined set of actions. However, it encourages the reuse of once defined actions by building libraries of action models which can be imported, composed and mapped to different web services. The impact, kind and target can be described in order to gain information on the composability of the action (cf. Section 3.3.1.3). Configuration properties are defined as a set of key-value-pairs to support all kinds of actions. NFComp has a real metamodel defined in Ecore which comes with strong tool support; it conforms to the standardized Meta Object Facility (MOF). Furthermore, there are different types of interdependencies that can be specified as additional constraints for defining valid groups of actions. For example, the *requires* interdependency defines that one action requires the use of another action, and the *mutex* interdependency prescribes that two actions cannot be used for the same functional point at once.

WS-Policy In WS-Policy [95] there is a coarse-grained concept of assertions which represent capabilities or requirements. There is no discrimination between fine-grained actions; e.g., there is only a single *EncryptedElements* assertion which is — depending on the role, for example, consumer or provider — executed either as encryption or decryption. There are no application constraints for the policies since there is no fine-grained action concept. In WS-Policy there is a standardized set of predefined actions. The different domain-specific specifications define concrete actions, e.g., WS-SecurityPolicy defines security-related actions such as encryption or signing. There is no impact description because the assertion concept is too general. An impact description does not make sense for all types of assertions (for instance, not for the ordering assertion *SignBeforeEncrypting*). Nonetheless, for example the *EncryptedElements* describes which elements are to be encrypted, which is similar to the impact target in NFComp. Configuration properties are similar to the impact description. There is no direct concept for configuration properties, but in theory arbitrary nested assertions can be used. The assertion concepts in WS-Policy are described rather sparsely and are not explicitly formalized in a metamodel. This makes it hard to define an abstract syntax for policies. XML Schema would be an appropriate abstract syntax. However, Heinzl et al. [41] showed that the definition of the abstract syntax in the specification is rather a textual informal description which defines how to use the different assertion types (by informally described examples). Redundant nested policy tags impede the usage of an XML Schema in order to validate the policy document. Thus it is complex to write

or read WS-Policy definitions and to provide tool support for this. Furthermore, there is no concept such as action interdependencies in the WS-Policy specification.

Ortiz and Hernandez In the work of Ortiz and Hernandez [68], actions (called extra-functional properties) are fine-grained, e.g., decryption and encryption are distinguished. There are no application constraints, though there is the *actionType* property defining if an action is executed before, after or instead of an operation. However, this is defined when applying the stereotype to an operation, i.e., when instantiating it. It is not possible to constrain the applicability (whether it is to be applied before or after) for a certain type of action. Predefined actions can be retrieved from a repository; however, it is not stated which actions can be used to produce fully generated code. Impact is not described. There are configuration properties for each action. They can be specified as properties of the UML stereotype for the respective action. The metamodel is based on UML and makes use of its profile mechanism. Action interdependencies are not supported at all.

Sec-MoSC In Sec-MoSC [88], rather coarse-grained actions are supported. It does not matter whether an action is assigned before or after a task, nor is messaging taken into account at all. There is no differentiation between *Encrypt* and *Decrypt*; both are summarized as *UseCryptography*. Hence, there are also no application constraints for actions. A set of ten actions is predefined (e.g., *UseCryptography*, *DeleteInformation*, *UseAccessControl*, and so on) for the nine predefined requirements, all for the security domain. An impact description is not possible, but support for configuration properties. The metamodel is based on Ecore. Action Interdependencies are not supported.

AO4BPEL In AO4BPEL [14], the action granularity is fine; e.g., encryption and decryption are distinguished and can be selected individually. There are no application constraints. For example, due to technical restrictions, atomic transactions can only be bound to *scope* activities. Moreover, some of the actions are implicitly (the appropriate advice type is generated depending on the requirement) applied to incoming or outgoing messages, for example encryption and decryption. For logging, the direction of the logging aspect can be defined by the respective class-type combination, e.g., *class=message*, *type=request*. There are predefined actions for security, reliable messaging, transactions and logging. There is no impact description. Configuration properties can be defined as child elements of a *requirement* element. XMLSchema allows validation of the deployment descriptor syntactically, but it is not possible to validate, for example, the support for *class*/*type* attribute combinations. Support is also lacking for any interdependencies between actions.

Summary Table 4.9 gives a summary of all discussed features and aggregates the feature points. NFComp received 6 feature points, supporting all features except the one of prede-

fined actions which is supported by AO4BPEL and WS-Policy. However, NFComp is the only approach which supports application constraints and action interdependencies.

Feature/Criterion	NFComp	Ortiz & Hernandez	WS-Policy	Sec-MoSC	AO4BPEL
Action granularity	Fine	Coarse	Fine	Coarse	Fine
Application constraints	Yes	No	No	No	No
Predefined actions	No	(Yes)	Yes	(Security)	Yes
Impact description	Yes	No	No	(Yes)	No
Configuration props.	Yes	No	(Yes)	Yes	Yes
Action metamodel	Ecore	UML /Ecore	(Schema)	Ecore	(Schema)
Action Interdeps.	Yes	No	No	No	No
Feature Points	6	1.5	3	3	3.5

Table 4.9: Feature Comparison in the Action Definition Phase

4.4.3 Action Composition

4.4.3.1 Case Study

NFComp Figure 4.11 presents the non-functional activities for the purchase order scenario. The model contains five non-functional activities. *DecryptVerify* and *SignEncrypt* describe the basic control flow for combining encryption and signatures and are used as nested activities. For example, *CacheSignEncrypt* extends *SignEncrypt* by the additional*WriteToCache* action.

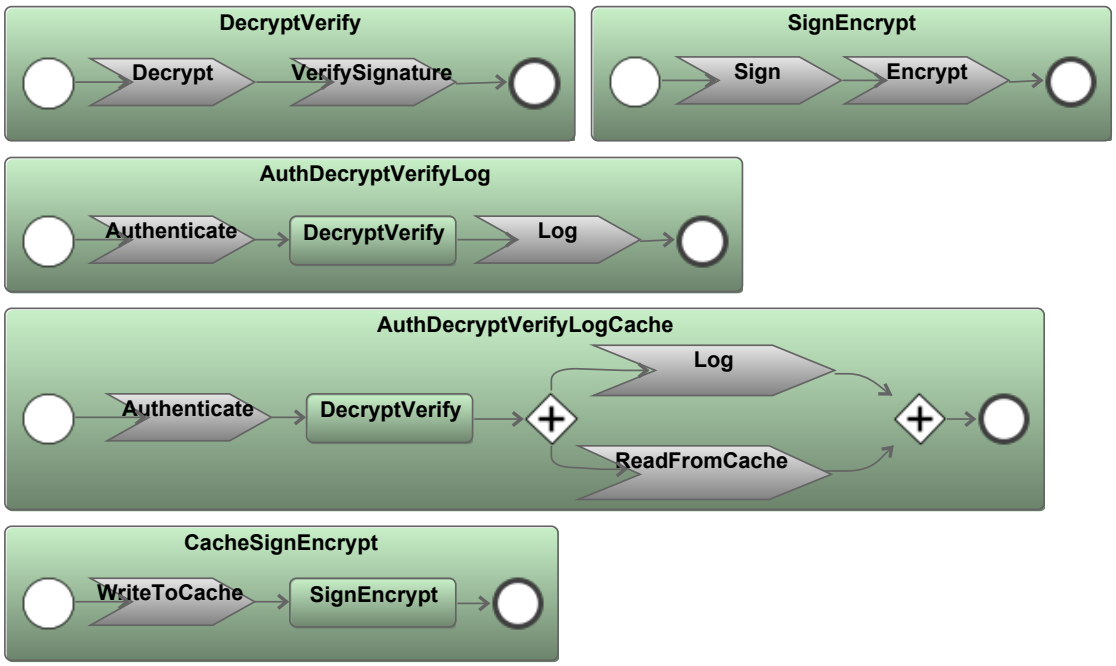

Figure 4.11: Non-Functional Activities for Purchase Order

Figure 4.12 shows the two composite non-functional actions for monitoring and reliable messaging. For in-order delivery of messages, a new reliable sequence is initiated using *StartRMSequence*. It follows at least one or arbitrary many *SendMessageInSequence* executions until *allMsgSent* is true. Then *TerminateRMSequence* is executed. The *TransactionalBehavior* activity is defined similarly. Using the composite action definition allows new interdependencies (all of *process* scope) to be derived based on the rules presented in Listing 3.4.

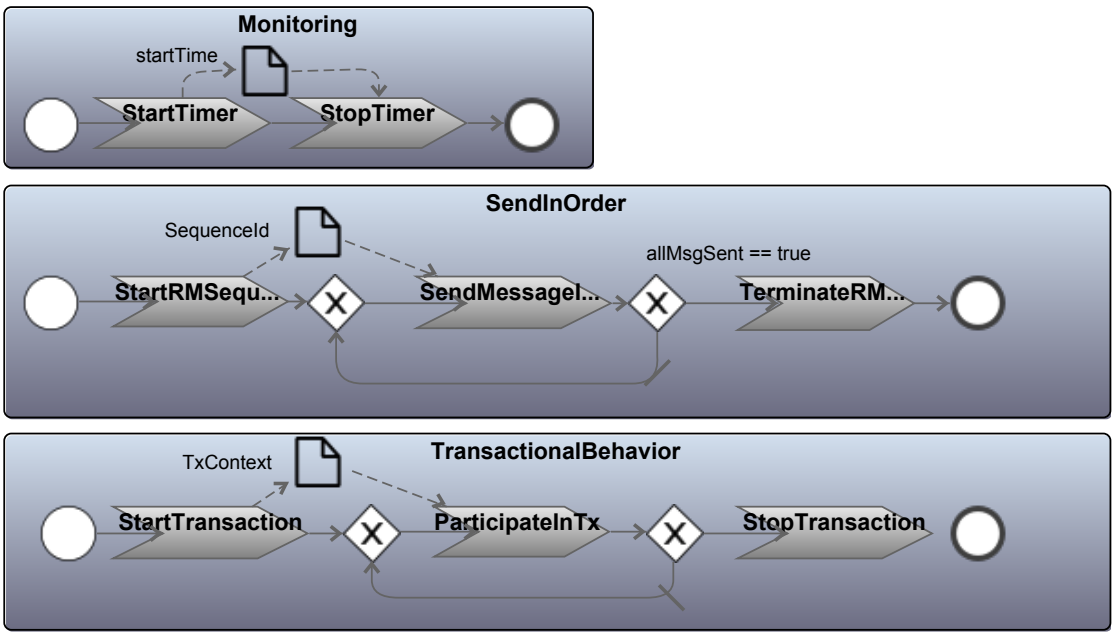

Figure 4.12: Composite Actions for Purchase Order

For illustration consider how the rules applied to the *SendMessageInOrder* activity. If *StartRMSequence* is defined as task *a* and *SendMessageInSequence* as *b*, then Rule 3 applies because there is an opening XOR gateway between them. Since there is no other exclusive branch into the same direction, Case 1 matches, introducing *precedes(StartRMSequence, SendMessageInSequence)*, *requires(StartRMSequence, SendMessageInSequence)* and *requires(SendMessageInSequence, StartRMSequence)*. In the next step *SendMessageInSequence* is defined as *a* and *TerminateRMSequence* as *b*. There is a closing XOR gateway between *a* and *b*, so Rule 2 applies and *precedes(SendMessageInSequence, TerminateRMSequence)*, *requires(SendMessageInSequence, TerminateRMSequence)* and *requires(TerminateRMSequence, SendMessageInSequence)* are introduced. There are two exclusive branches, though there is only one action in the branch, and the branch is in another direction (backwards, forwards). This is why Rule 4 does not apply. The backwards branch closing the loop does not produce new interdependencies because *a* = SendMessageInSequence = *b*. Figure 4.13 shows the resulting process-scoped interdependencies (for brevity, transactional actions have been omitted).

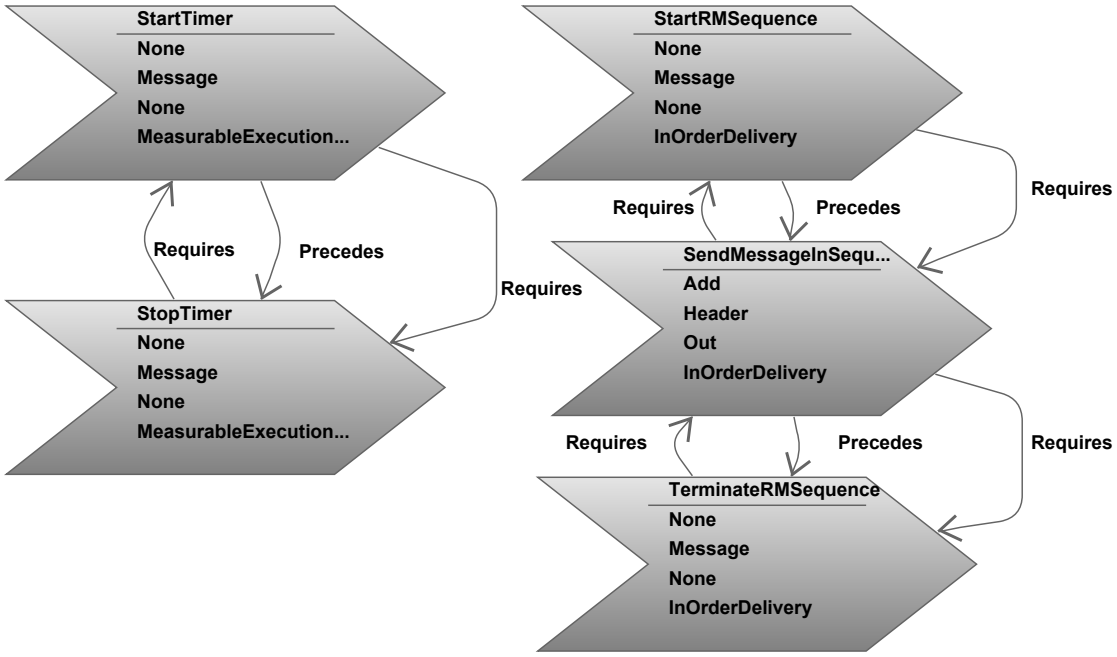

Figure 4.13: Interdependencies Inferred from Composite Actions

WS-Policy In WS-Policy [95], groups of actions can be defined using combinations of the *ExactlyOne* and *All* elements. In Listing 4.5 it has been defined that either only reliable messaging or reliable messaging plus encryption can be used. If this policy has been defined for a service, the consumer will be able to choose one of these assertion groups. If the consumer defines a policy herself, the policy can be intersected with the one of the service in order to calculate the effective options to be used. The role of the *All* element is to define that all of the assertions must be guaranteed, though *All* does not define any ordering restrictions.

```
1  <wsp:Policy>
2    <wsp:ExactlyOne>
3      <wsp:All>
4        <wssp:EncryptedElements>
5          <wssp:XPath>/S:Envelope/S:Body</wssp:XPath>
6        </wssp:SignedElements>
7        <wssp:EncryptedElements>
8          <wsssp:XPath>/S:Envelope/S:Body</wssp:XPath>
9        </wssp:EncryptedElements>
10     </wsp:All>
11     <wsp:All>
12       <wsrmp:RMAssertion/>
13     </wsp:All>
14   </wsp:ExactlyOne>
15 </wsp:Policy>
```

Listing 4.5: WS-Policy Defining Different Groups of Assertions

With respect to the ordering of actions/assertions, the WS-Policy specification states that there is no general ordering mechanism defined, but domain-specific specifications may

define their own assertions with ordering semantics. This strategy has been followed in WS-SecurityPolicy. Listing 4.6 shows an application of the *SignBeforeEncrypting* ordering assertion. Alternatively, the *EncryptBeforeSigning* assertion could have been used. There are no further ordering assertions defined in specifications such as WS-SecurityPolicy, WS-Addressing or WS-ReliableMessagingPolicy.

```
<wsp:Policy>
  <sp:SymmetricBinding>
    <wsp:Policy>
      <sp:ProtectionToken>
        <wsp:Policy>
          <sp:Kerberos sp:IncludeToken=".../IncludeToken/Once" />
            <wsp:Policy><sp:WSSKerberosV5ApReqToken11/></wsp:Policy>
          </sp:Kerberos>
        </wsp:Policy>
      </sp:ProtectionToken>
      <sp:SignBeforeEncrypting />
      <sp:EncryptSignature />
    </wsp:Policy>
  </sp:SymmetricBinding>
  <sp:SignedParts>
    <sp:Body/>
  </sp:SignedParts>
</wsp:Policy>
```

Listing 4.6: Security Policy Using Ordering Assertion

Ortiz and Hernandez In the approach of Ortiz and Hernandez [68], a priority-based ordering concept is followed. Each extra-functional property may define its priority. The priority defines the order of the extra-functional properties when applied to the same target. The priority is a property of a stereotype and its value is defined when applying the stereotype to its target, e.g., an operation or an interface. The priority is enforced in generated aspects by the use of AspectJ's precedence mechanism (more details can be found in Section 4.4.5).

Sec-MoSC In Sec-MoSC [88] there is a grouping concept for actions called *NF-Groups*. An *NF-Group* is graphically represented by an expandable box and can be associated with an *NF-Attribute*. It is however not clear if an *NF-Group* defines any ordering restrictions upon the grouped actions. Figure 4.14 depicts the purchase order process with *NF-Attributes* and the associated *NF-Groups* of *NF-Actions* for the purchase order service.

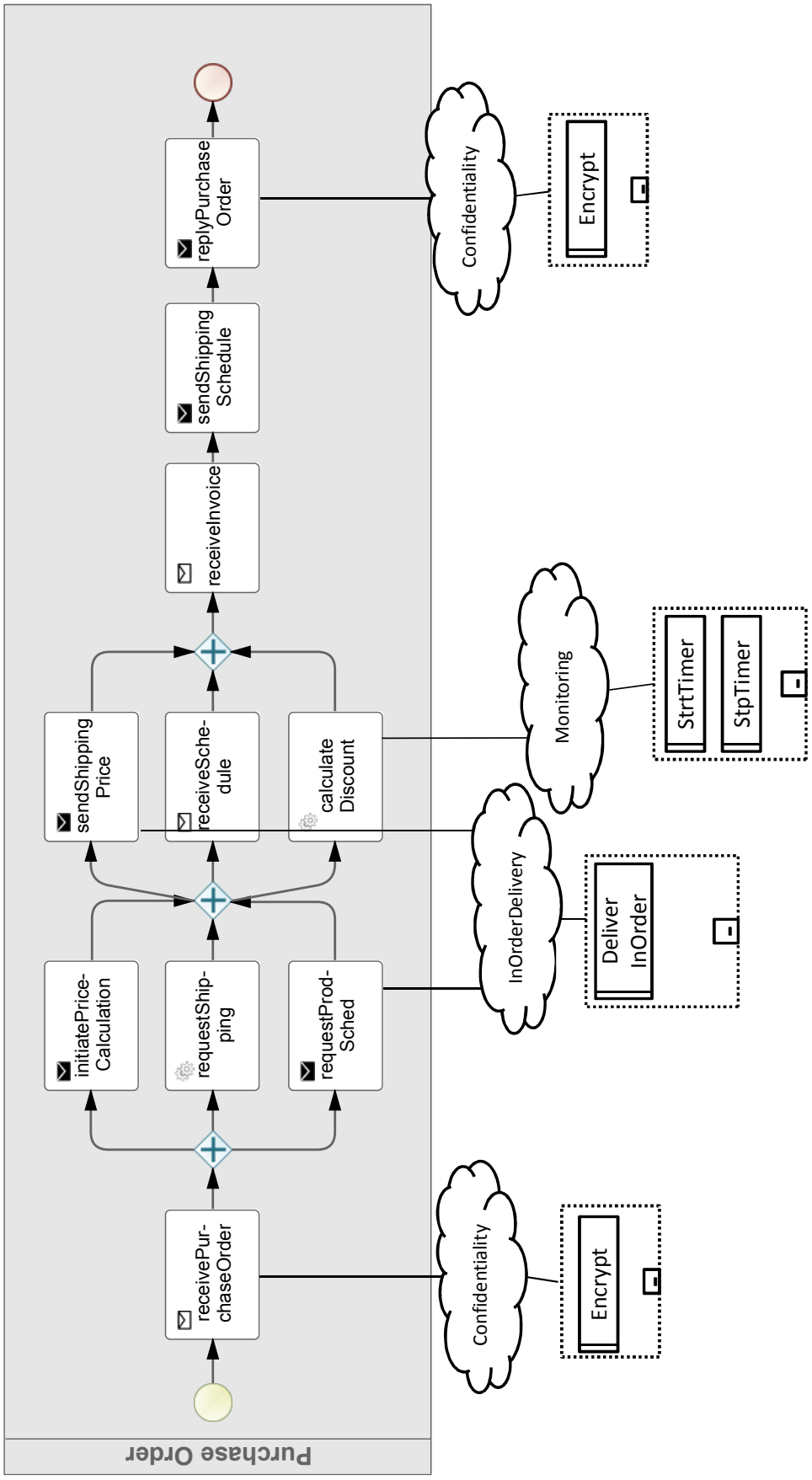

Figure 4.14: Groups of Actions for the Purchase Order Process with Sec-MoSC

AO4BPEL AO4BPEL [14] uses the concept of priorities. This concept is applied on the aspect level to define the ordering of aspects which match for the same join point and use the same advice type. There is a natural order by advice type: *before*, *around*, *before soapmessageout*, *around soapmessageout*, *after soapmessagein* and *after*. Whenever two advice elements of the same type apply to the same join point, the one with the lower priority is executed first. Additionally, there are priority placeholders for the first and last advice called *first* and *last*. This is because particular aspects — due to technical reasons — always need to be the first or the last to be executed. For example, the reliable messaging service sends the message out, and thereafter no further aspects can match. Hence, aspects calling particular operations of the reliable messaging service need to be the last ones to be executed. AO4BPEL uses the deployment descriptor to declaratively describe the non-functional requirements and to generate aspects, automatically. Hence, the priority attribute is part of the *requirement* element of the deployment descriptor. Listing 4.7 demonstrates the use of the priority attribute for different requirements. This configuration defines that *username authentication* must be executed before *decrypt*, which must be executed before *checkSignature*.

```
<bpel-dd>
  ...
 <services>
  <service name="reliablemessaging">
   <requirements>
    <requirement name="req0" class="semantics"
                 type="inOrder" priority="last" />
   </requirements>
  </service>
  <service name="security">
   <requirements>
     <requirement name="req1" class=" integrity " priority="0"
                  type="sign">
     </requirement>
     <requirement name="req2" class=" confidentiality " priority="1"
                  type="encryption">
     </requirement>
     <requirement name="req3" class=" authentication " priority="0"
                 type="usernametoken" />
     <requirement name="req4" class=" confidentiality " priority="1"
                  type="decrypt">
     </requirement>
     <requirement name="req5" class=" integrity " priority="2"
                  type="checkSignature">
     </requirement>
    </requirements>
   </service>
  </services>
</bpel-dd>
```

Listing 4.7: Priorities in AO4BPEL Deployment Descriptor

4.4.3.2 Feature Comparison

Criteria During the implementation of the purchase order process with NFComp and other related approaches, a set of features with respect to action composition has been collected. This feature set is used in the following to compare the approaches with each other. Firstly, it is important to be able to group related actions in order to map this group to a single functional point. Executing this group means executing all contained actions for the same target subject. Secondly, it is important to define an execution order for all contained actions. For ordering, the main features are: sequential order, support for parallelism and exclusive branches (xor semantics). However, data flow specification is also important to pass data between particular actions. For example for monitoring, the *StartTimer* action may pass the captured starting time to the *StopTimer* action. Another important feature is to validate action composition, which is in fact necessary because of the variety of actions from different non-functional domains. As for requirements and actions, a predefined set of non-functional activities or composite actions makes sense. This allows making use of best practices provided by the approach; e.g., an approach could suggest how to combine caching with encryption in general. Moreover, the support for composite actions is an important feature for making it possible to provide a fine-grained mapping of atomic actions. These actions should be mapped to the process in the order according to the composite action's process definition. Another important feature is to define orderings specific to a particular service or even a particular operation of this service. For example, in the case of encryption and caching, particular services may have strict security requirements and messages must be encrypted even when stored in the cache and other services not. Hence, for the former type of service *Encrypt* must be executed before *WriteToCache* and for the latter type it is sufficient to execute *Encrypt* after caching.

NFComp In NFComp, the non-functional activity concept defines both a container for groups of actions and an ordering mechanism. If there is no ordering restriction at all, the contained actions can be modeled as completely parallel. Sequential ordering is possible via sequence flow connections. Exclusive ordering can be defined via exclusive XOR gateways, and parallel ordering, through parallel AND gateways. Dataflow can be modeled with data items and data associations. The composition can be validated against the interdependencies defined during the Action Definition Phase. A composite action can be used to model — similar to non-functional activities — the process of actions across several process steps. An example is the reliable messaging sequence, which involves different atomic actions such as reliable sequence creation, participation and cancellation. NComp supports service-specific orderings because for each service distinct non-functional activities can be mapped. In the purchase order scenario, this could, for example, make sense for the ordering of the *WriteToCache* and *Encrypt* actions. However, to be flexible enough in this case, the *precedes* interdependency between them should be removed. Otherwise, the validation mechanism would discover that an activity executing *Encrypt* before *WriteToCache* would violate the interdependency constraints.

WS-Policy The focus of WS-Policy [95] is on interoperability and policy negotiation (through normal forms and policy intersection) rather than on action ordering. This weak ordering support in WS-Policy leads to assertions which reflect orderings. The problem with such an ordering assertion is that for each combination of assertions, a new ordering assertion must be invented. This does not scale well with the number of assertions to be ordered; e.g., if an ordering among three assertions should be defined, 6 ordering assertions already must be introduced. To define orderings between assertions across different domains makes the situation even more complex because the domain-specific specifications must be aware of each other. It is possible to define groups of actions using the *All* element. Composite actions are not supported; at least no horizontal ordering restrictions can be defined. There are no predefined sets of actions, nor can dataflow be described. A validation mechanism is not available. Since WS-Policy does not support orderings appropriately a service-specific ordering is not supported.

Ortiz and Hernandez In the approach of Ortiz and Hernandez [68], priorities can be used to define the ordering between actions (extra-functional properties). However, a priority-based approach is not sufficient. Firstly, it is not flexible enough, since it is impossible to insert a new priority between two subsequent priorities without changing the priorities of other properties. Secondly, only sequential orderings can be specified. The specification of dataflow is not supported to be specified as well as predefined compositions and composite actions. Properties of stereotypes are instantiated when the stereotype is applied to its target. Hence, the priority attribute can be set for each service/operation individually. In general, there are limitations with AspectJ which is used for the realization/enforcement. In AspectJ a precedence order is defined in an aspect and is not specific to a particular join point. However, in [68] a new aspect for each stereotype application is generated and thus a precedence order is always defined per particular stereotype application and not per stereotype.

Sec-MoSC Sec-MoSC [88] offers composition via *NF-Groups*. It is not clear whether there is a composition order, but if so, it is not more than sequential because there are no ordering concepts in the model (0.5 feature points for sequential ordering) and probably predefined and not flexible. The binding between *NF-Groups* and *NF-Attributes* is possibly unsuitable. If there is an adequate breakdown of the requirements into atomic *NF-Attributes*, there will (usually) be a 1-1 association between attributes and actions. It is more common, then, to bind multiple attributes to the same task. This makes the *NF-Groups* concept somehow obsolete. Alternatively, more abstract composite attributes need to be mapped to process tasks. The order of actions is not completely flexible at runtime because Axis2 Handler Phases are used, including the Rampart module and a self-designed MoSC Security Module. This cannot be integrated between fine-grained security actions since Rampart uses coarse grained handlers (*SecurityIn*, *SecurityOutHandler* plus policy-based ones). In fact, this is another hint for a given, predefined order of actions. There are no predefined, reusable compositions of actions, at least no apparent ones. The execution order is probably given by the Axis2 modules being used. It is not clear

if there is a ordering within *NF-Groups*, but if so, at least activity-specific (and some activities will actually invoke services, hence 0.5 feature points) ordering would be possible.

AO4BPEL AO4BPEL [14] does not support groups of actions. Each requirement is specified individually, and there is only one predefined grouping which is by service, e.g., reliability or security. The ordering of actions can be influenced using the priority attribute for a requirement. Still, there are two limitations. Firstly, particular requirements produce several aspects of potentially different priorities. It is not clear how to calculate the priority for these composite requirements. Secondly, particular requirements are not flexible enough to be prioritized as desired. For example, reliable messaging must be the last action to be executed. Moreover, there is a natural advice-type-based order which is taken into account before priority is processed. Depending on the advice types being generated, this may lead to limitations. If *username authentication* had a lower priority than *decryption*, then *decryption* would be executed first. A reason for this prioritization could be that the whole message including its header has been encrypted and should be transformed into plain XML before processing the username token. Given this example and presuming that *decryption* generates an advice of type *after soapmessagein* and *username authentication* one of type *around soapmessageout*, this would lead to an obvious limitation. Although the priorities are clearly set, *username authentication* is executed before *decryption* because *around soapmessageout* is always processed before *after soapmessagein*. Dataflow specification is not supported, nor can compositions be validated. Composite actions are not visible in AO4BPEL; e.g., the reliable messaging sequence comprising three individual actions is represented by a single requirement of type *inOrder*. This requirement is usually applied to a structured BPEL activity such as a *sequence* or *scope*, meaning that all messaging activities will be invoked with the *inOrder* delivery assurance. It is, however, not possible to select only a subset of messaging activities in this *sequence* with the delivery assurance. This argument is also valid for transactions which can be applied only on a structured activity level. In AO4BPEL, service-specific orderings can be defined, because for each requirement in the deployment descriptor a new aspect is generated. However, requirements must be duplicated and priorities must be redefined to do so.

Summary Table 4.10 summarizes the discussed features and aggregates the feature points. NFComp received 8 of 9 possible feature points. It does not offer predefined compositions; however, this feature is not supported by any of the compared approaches, either.

Feature/Criterion	NFComp	WS-Policy	Ortiz & Hernandez	Sec-MoSC	AO4BPEL
Groups of Actions	Yes	Yes	No	Yes	No
Sequential Ordering	Yes	No	Yes	No?	(Yes)
Controlflow Parallelism	Yes	No	No	No	No
Controlflow Exclusiveness	Yes	No	No	No	No
Dataflow	Yes	No	No	No	No
Composition Validation	Yes	No	No	No	No
Predefined Compositions	No	No	No	No	No
Composite Actions	Yes	No	No	No	No
Service-specific Ordering	Yes	No	Yes	(Yes)	Yes
Feature Points	8	1	2	1.5	1.5

Table 4.10: Feature Comparison in the Action Composition Phase

4.4.4 Action to Application Mapping

4.4.4.1 Case Study

In the next phase, the actions and sets of actions need to be mapped to the target components or subjects. The process of mapping depends generally on the view. In the black box view, the mapping is done on service or operation level. In the gray box view actions are mapped to process tasks, and in the white box view they are mapped to internal methods implemented in a particular programming language. In the following, the actions are assigned, depending on the requirements, to their target subjects in the purchase order service. The black box mapping is defined as follows:

- **Purchase Order Service** *Incoming*: startTimer *Outgoing*: stopTimer
- **sendInvoice** *Incoming*: verifySignature, decrypt *Outgoing*: encrypt, sign
- **sendPurchaseOrder** *Incoming*: log, verifySignature, decrypt, authenticate *Outgoing*: encrypt, sign
- **getAvailableItemsByCategory** *Incoming*: log, verifySignature, decrypt, authenticate, readFromCache *Outgoing*: writeToCache, encrypt, sign

The gray box mapping is defined as:

- **requestProductionSchedule Task** *Incoming*: log *Outgoing*: sendMessageInSequence, log
- **sendShippingPrice** *Incoming*: log *Outgoing*: sendMessageInSequence, log
- **calculateDiscount** *Incoming*: startTimer, log *Outgoing*: stopTimer, log
- **initiatePriceCalculation** *Incoming*: log *Outgoing*:log

- **requestShipping** *Incoming*: log *Outgoing*:log
- **receivePurchaseOrder** *Incoming*: log *Outgoing*:log
- **replyPurchaseOrder** *Incoming*: log *Outgoing*:log
- **receiveSchedule** *Incoming*: log *Outgoing*:log
- **receiveInvoice** *Incoming*: log *Outgoing*:log
- **sendShippingSchedule** *Incoming*: log *Outgoing*:log

The white box mapping is:

- **createNewCategory** *Before Method*: startTransaction
- **executeUpdate** *Around Method*: participateInTx
- **createNewCategory** *After Method*: stopTransaction

NFComp In NFComp, black box mapping is done in a graphical way. Firstly, the mapping editor is used to import the WSDL of the purchase order service. A new service box appears in the palette, which can be dragged into the diagram. The box can be marked, and it is possible to drill down to the operation level (and to navigate back, clicking on the ServiceRef symbol). The operation level view is shown in Figure 4.15. The activities from the action composition phase have been mapped to the incoming, respectively outgoing, messages of the operations according to the above-defined mapping.

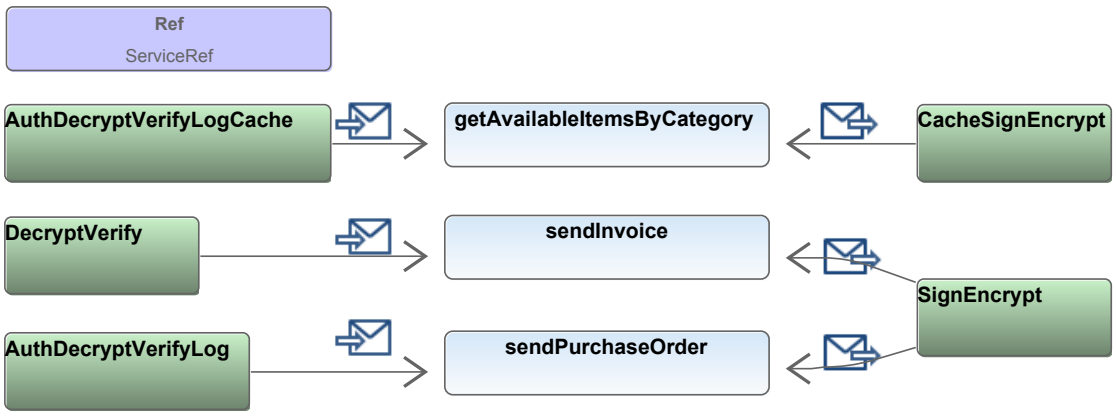

Figure 4.15: Mapping of Actions to Purchase Order Service (Black Box View)

In the gray box view, the BPMN2 definition of the purchase order service is imported. The process is presented graphically in the mapping editor. Then, the actions defined in the previous phases can be mapped to the process tasks. Figure 4.16 shows the mapping of the purchase order process according to the above-defined mapping. The editor allows only valid association types; e.g., it is not allowed to draw an outgoing message association to a *ReceiveTask*. The interdependencies can be validated by using the validation button (not shown in the figure). In case of a problem, the list of errors is shown in the errors view. In black box mapping, interaction and invocation scope interdependencies are taken into account, whereas in the gray box view, all three types of scopes are validated.

The mapping in the white box view is similar to the one in the gray box view, however, the view reveals also behavior from internal Java classes in form of method invocations. The behavior is represented as a BPMN2 diagram and can be reverse-engineered from existing Java code. This allows to import the generated BPMN2 diagram file in the mapping editor. Figure 4.17 shows the resulting mapping for purchase order for the methods *getAllItemsByCategory* and *createNewCategory*. The monitoring and transaction actions have been mapped using the well-known association types, whereby the association with direction type *In_Out* is used with *around* semantics. In this mapping, monitoring actions are mapped to all methods which access the database for reading data.

The monitoring actions are examples for multi-instance actions described in Section 3.5.4. The *StartTimer* and *StopTimer* actions are mapped twice and the monitoring composite action manages state in form of the *startTime* data item (see Figure 4.12). Hence, the properties sheet can be used to set the *instanceIdentifier* of the associations. The *StartTimer* association on the left and the *StopTimer* association on the left will be set to value *instA* and the associations on the right are assigned to *instB*. This identifies the pairs of association belonging to the same monitoring instance. Actions for transactions are mapped before, respectively after, the *createNewCategory* method. The *executeUpdate* method writes to the data base and is invoked during the execution of *createNewCategory*, shown in the pool below. Thus, the *participateInTx* action has been mapped with type *around* in order to execute the method in the context of a transaction with all-or-nothing semantics and isolation.

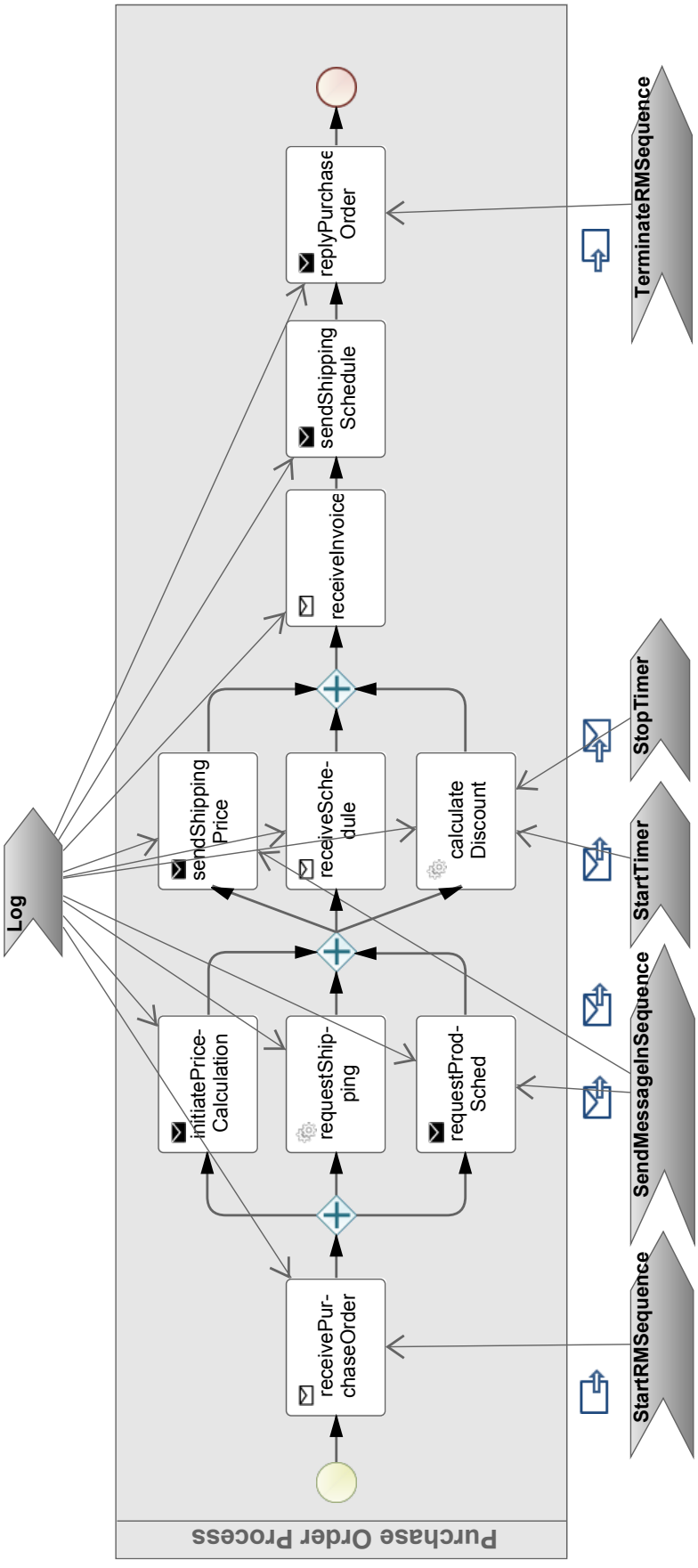

Figure 4.16: Mapping of Actions to Purchase Order Service (Gray Box View)

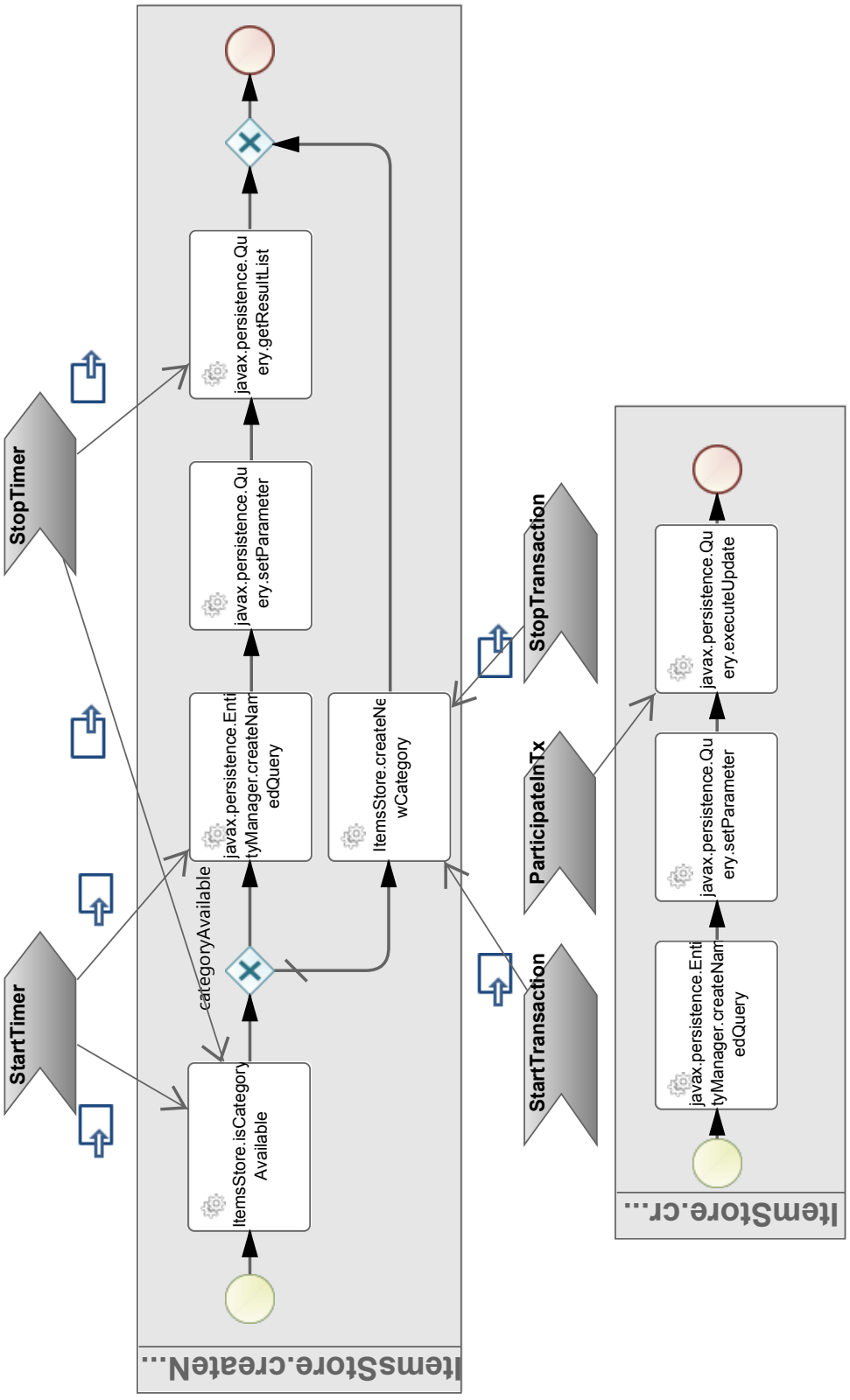

Figure 4.17: Mapping of Actions to Purchase Order Service (White Box View)

Ortiz and Hernandez In the approach of Ortiz and Hernandez [68], the mapping is represented by UML Stereotype application to interfaces and operations. Firstly, a new UML Profile must be applied to the UML Model. Then, the different UML Stereotypes can be applied to the interface/operations representing the purchase order service from the black box view. This stereotype application instantiates the stereotype and all its properties. Thus, for each application, the different properties such as *actionType*, *policyId* and so on must be specified. Figure 4.18 depicts the purchase order service interface annotated with the stereotypes according to the above-defined mapping.

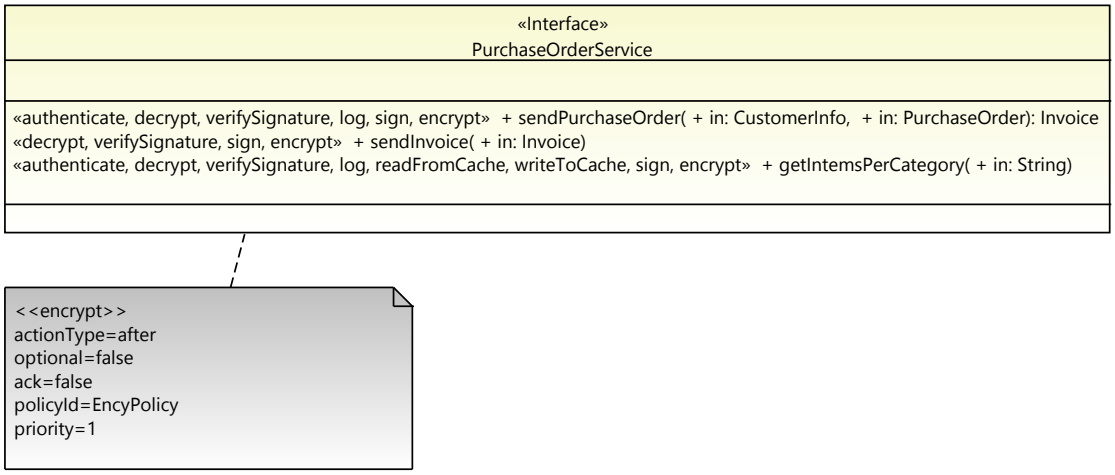

Figure 4.18: Mapping of Extra-Functional Properties to Purchase Order Service in [68]

Sec-MoSC In Sec-MoSC [88] there is also a graphical mapping. *NF-Bind* connections are used to connect BPMN process tasks with *NF-Attributes*. There is no direct connection from process tasks to *NF-Actions*. They are connected via *NF-Attributes*. Not only can individual actions be mapped, but groups of actions (*NF-Groups*) can as well. The mapping for the purchase order scenario can be seen in Figure 4.14. Each *NF-Group* (containing *NF-Actions*) is mapped to an *NF-Attribute* which is connected to a process task.

WS-Policy In WS-Policy [95], the mapping of policies to target subjects is defined in the WS-PolicyAttachment [96] specification. There are two ways to attach a policy to its subject. The first way is to use a dedicated XML element directly in the target subject, e.g., the WSDL definition, which points to a policy using an URI. The second way is to use an external policy attachment, which is not included in the target subject. In the external attachment, there is an *AppliesTo* element describing to which subject the attachment applies and a *PolicyReference* element pointing to the policy. Listing 4.8 shows the external policy attachment from the *POPolicy* defined in Listing 4.4. The mapping is based on the *EndpointReference* to the *purchaseOrderPT* port type and service named *PurchaseOrderService*. In the first variant of policy attachments, the policy reference can directly be included in one of the following WSDL elements: *service*,

port, *binding*, *portType*, *operation*, *input*, *output*, *fault* and *message*. The WS-PolicyAttachment [96] specification supports only the black box view based on the service interface. Charfi et al. [11] extended this specification by the feature of selecting WS-BPEL elements instead of WSDL elements. With this extension WS-Policy can also be used for gray box mapping.

```
<wsp:PolicyAttachment>
  <wsp:AppliesTo>
    <wsa:EndpointReference xmlns:myws="..." >
      <wsa:Address>http://myws.example.com/PurchaseOrder</wsa:Address>
      <wsa:PortType>myws:purchaseOrderPT</wsa:PortType>
      <wsa:ServiceName>myws:PurchaseOrderService</wsa:ServiceName>
    </wsa:EndpointReference>
  </wsp:AppliesTo>
  <wsp:PolicyReference
    URI="http://myws.example.com/policies#POPolicy" />
</wsp:PolicyAttachment>
```

Listing 4.8: WS-Policy Attachment

AO4BPEL AO4BPEL [14] does not provide a graphical mapping. Instead, the XPath [19] expression language can be used to define the mapping between requirements and WS-BPEL process elements. Listing 4.9 extends the previously introduced deployment descriptor (see Listing 4.7) by so-called selectors. Selectors are used to connect requirements with WS-BPEL process elements. Each selector defines an XPath expression to select the element(s) and an identifier which is used to connect requirements with a particular selector (*selectorid* attribute). In this listing, the mapping of reliable messaging and logging is shown exemplarily. The mapping of the reliable messaging *inOrder* requirement is set to the main sequence that spans all activities of the WS-BPEL process. This is due to a limitation of AO4BPEL. There are two aspects generated out of this requirement: one applying to a structured activity and one applying to a messaging activity. Thus, a structured activity always must be identified with a selector. The second aspect applying to messaging activities will add an *//invoke* XPath expression to the pointcut expression generated out of the selector. This will cause all *invoke* messaging activities to participate in the reliable messaging sequence. This is, though, not intended for the purchase order process. The problem can only be solved by restructuring the WS-BPEL process.

```
<bpel-dd>
  <selectors>
    <selector id="0" name="mainsequence" type="activity">
      /process[@name="PurchaseOrder"]/sequence[@name="mainsequence"]
    </selector>
    <selector id="1" name="allMsgActivities" type="activity">
      /process[@name="PurchaseOrder"]//invoke | /process[@name="PurchaseOrder"]//receive | /process[@name="
          PurchaseOrder"]//reply
    </selector>
  </selectors>
  <services>
    <service name="reliablemessaging">
      <requirements>
```

```
13          <requirement name="req0" class="semantics" selectorid="0"
14            type="inOrder"/>
15        </requirements>
16      </service>
17       ...
18      <service name="logging">
19        <requirements>
20          <requirement name="req6" class="message" selectorid="1"
21            type="request">
22          </requirement>
23          <requirement name="req7" class="message" selectorid="1"
24            type="response">
25          </requirement>
26        </requirements>
27      </service>
28    </services>
29  </bpel-dd>
```

Listing 4.9: Selectors in AO4BPEL Deployment Descriptor

4.4.4.2 Feature Comparison

Criteria In the following, a set of features and criteria have been chosen to provide a feature comparison for the relevant approaches in this phase. The first criterion is the type of mapping language used to connect non-functional actions/properties with the functional subjects. There are different approaches. Firstly, there are graphical approaches versus textual ones. Secondly, there are direct mappings; i.e., the action/property is directly integrated into the target artifact, for example, into the WSDL or BPMN specification; and association-based ones, i.e., externally attached to the target artifacts. The latter approach is more complex, but allows to strictly separate non-functional from functional concerns. Both approaches have benefits and drawbacks, hence they are not rated in terms of feature points. The next features to be compared are the different views on web services — black, gray and white box — which are supported by the respective approach. The validation of mappings is also an important feature, because it is quite complex, for example, to map composite actions to composite web services. In this mapping, the composition logic of composite non-functional actions, non-functional activities and composite web services must be taken into account. Furthermore, it is important to be able to map groups of related actions to one and the same subject. Sometimes the behavior of a non-functional action may be specific to a certain mapping. In this case it should be possible to adapt the behavior by, for example, changing/overriding the configuration. A composite action mapping, i.e., mapping individual actions pertaining to a composite non-functional action, should also be possible, because it is much more flexible than mapping the composite action at once.

NFComp NFComp uses graphical association concepts based on connection lines to map actions to their target subjects. It supports black box, gray box and white box mapping. The

mapping can be validated against the interdependency model, in which even different scopes are taken into account. Groups of actions (non-functional activity) as well as individual actions can be mapped. It is possible to configure action configuration properties based on the association; i.e., the default configuration defined on the action level can be overwritten for a specific mapping. Composite actions are supported to be mapped. Their composition logic can be defined in the action composition phase. This logic can be translated into interdependencies in *process* scope. These additional interdependencies allow validation as to whether the mapping is correct according to the composition logic of the composite action.

Ortiz and Hernandez In the approach of Ortiz and Hernandez [68], extra-functional properties are directly mapped to operations or interfaces. Since interfaces and operations are the only target subjects, only the black box view is supported. The mapping cannot be validated. Through the direct mapping, it is possible to assign different properties to actions based on the application of stereotypes to interfaces/operations. This is possible because each stereotype is newly instantiated when applied to its target. Individual actions pertaining to composite actions cannot be mapped individually, though.

Sec-MoSC In Sec-MoSC [88], there is also — similar to NFComp — a graphical mapping based on connection lines, though, between *NF-Attributes* and process tasks. Sec-MoSC supports gray box mapping only since the mapping is based on BPMN processes. The mapping cannot be validated. There is an *NF-Group* concept to map groups of actions to an *NF-Attribute*. A mapping-specific configuration of actions is not possible; neither is the mapping of actions pertaining to composite actions.

WS-Policy With WS-Policy [95], policies can directly be attached to their target subjects. However, an external attachment is also possible via endpoint references (EPRef for short). The WSDL interface of a service is the target subject and thus it supports the black box view. In applying the approach of Charfi et al. [11] the gray box view is also supported (because this is not out-of-the box it is rated only with 0.5 feature points). A validation of the mapping is not supported. Mapping-specific configurations are not directly possible. If so, the policy definitions would have to be duplicated. Also, the mapping of individual actions pertaining to a composite action is not supported.

AO4BPEL In AO4BPEL [14], the mapping is performed using the XPath expression language. AO4BPEL supports only the gray box view because it aims at WS-BPEL processes. Groups of actions cannot be mapped at once and a validation of the mapping is not possible. A mapping-specific configuration is also not supported. Composite actions are treated as atomic ones in the deployment descriptor. This is why atomic actions pertaining to a composite action cannot be mapped individually.

Summary In summary, there are benefits and drawbacks with graphical mappings compared to textual expression languages. In the graphical mappings presented in this section (NFComp and Sec-MoSC), there is an explicit direct mapping between different elements. In the textual expression language, there is rather a kind of query selecting from a set of elements. In fact, this is much more powerful because it allows the selection of a whole set of elements at once (for example, it is useful to map the *Log* Action to all purchase order tasks). It is, though, more complex to determine the elements selected by the query. If each element is explicitly connected to its target, this will be more human readable. Another argument is the different behavior with respect to changes in the mapping targets. If, for example, a process task is removed from the purchase order process, an error will occur in the explicit mapping when an action has been mapped to this very task. In the expression language-based approach, there is no such error. The mapping still works, but the mapped action will never be executed.

Table 4.11 summarizes the discussed features and aggregates the feature points. Features which have not been rated use an italic font type. NFComp received 7 of 7 possible feature points. It is the only approach supporting white box mapping, mapping validation, mapping-specific configurations and composite action mapping.

Feature/Criterion	**NFComp**	**Ortiz & Hernandez**	**Sec-MoSC**	**WS-Policy**	**AO4BPEL**
Mapping Language	Assocs	Direct	Assocs	Direct/ EPRef	XPath
Black Box Mapping	Yes	Yes	No	Yes	No
Gray Box Mapping	Yes	No	Yes	(Yes)[11]	Yes
White Box Mapping	Yes	No	No	No	No
Mapping Validation	Yes	No	No	No	No
Mapping Groups of Actions	Yes	No	Yes	Yes	No
Mapping-Specific Config	Yes	No	No	No	No
Composite Action Mapping	Yes	No	No	No	No
Feature Points	7	1	2	2.5	1

Table 4.11: Feature Comparison in the Mapping Phase

4.4.5 Middleware Mapping/Code Generation

4.4.5.1 Case Study

In the *Code Generation Phase*, the modeled action composition is transformed into executable code. This is done in order to enforce the modeled action composition at runtime.

NFComp In NFComp, it is assumed that middleware services take over the responsibility of realizing the non-functional actions. Each middleware service operation implements one non-functional action. If multiple operations need to be called to realize a single action, this action

should be modeled as a composite action. Middleware services in NFComp are either local mediators (and thus specific to the enterprise service bus being used), web services or Java classes (white box view). Web services are more flexible, reusable and easily integrated into NFComp. The drawback of middleware web services is the messaging overhead and the problem that they can hardly be used to realize end-to-end middleware scenarios such as required for security or reliable messaging. If web services are being used for this purpose, it must be ensured that they are accessible in a secure and reliable manner. For example, it does not make sense to use a security web service outside the corporate network over an insecure channel.

The following non-functional actions for the purchase order scenario have been realized as web services:

- **Log** has been realized with the *LoggingWebService* called by the *RemoteMiddlewareServiceMediator*
- **Authenticate** has been realized with the *AuthWebService* called by the *RemoteMiddlewareServiceMediator*

The following non-functional actions for purchase order have been realized as Synapse mediators:

- **ReadFromCache** and **WriteToCache** has been realized with the *CachingMediator*
- **Sign** has been realized with the *SignMediator*
- **VerifySignature** has been realized with the *VerifySignatureMediator*
- **Encrypt** has been realized with the *EncryptMediator*
- **Decrypt** has been realized with the *DecryptMediator*
- **StartTimer** has been realized with the *StartTimerMediator*
- **StopTimer** has been realized with the *StopTimerMediator*
- **StartRMSequence** has been realized with the *StartRMSequenceMediator*
- **SendInOrder** has been realized with the *SendInOrderMediator*
- **TerminateRMSequence** has been realized with the *TerminateRMSequenceMediator*

The following non-functional actions have been realized as plain Java classes:

- **StartTimer** *startTimer* method in class *WSMonitoring*
- **StopTimer** *stopTimer* method in class *WSMonitoring*
- **StartTransaction** *start* method in class *WSTransaction*

- **ParticipateInTx** *participate* method in class *WSTransaction*
- **StopTransaction** *end* method in class *WSTransaction*

Middleware mapping for the *LoggingWebService* is performed with the action definition editor. The WSDL is imported into the editor, making the operations of the web service available for selection in the mapping table. This mapping table is shown in the properties view when selecting a particular action. In the case of logging, the *log* operation of the *LoggingWebService* is chosen. Additionally, key-value-pairs can be mapped. In this case, the key *message* (corresponding to the message parameter of the *log* operation) is set to the variable *$message*. The *logLevel* key (corresponding to the *logLevel* parameter of the *log* operation) is set to the constant value *info*.

In the case of local mediators being used as implementation for non-functional actions, no operation is selected, but instead, the *localName* parameter of the mapping is set to the name of the mediator, e.g., *encrypt* in case of the *EncryptMediator*. When the mapping of all non-functional actions to middleware services is defined, the code generation in NFComp can be started.

In the following, the mapping of Java classes as middleware service implementation is described. In this description the *participateInTx* action is used as an example. This action is responsible for executing a particular functionality in the context of a transaction. Firstly, the *localName* middleware mapping attribute is set to *WSTransaction* which is the class name of the middleware service (package name has been omitted for brevity). Secondly, the *operation* middleware mapping attribute is set to *participate*. Thirdly, the *type* attribute is set to *reflective* for the *participateInTx* action because, the *participate* method will execute the join point itself in order to do this in a transactional context. This is, for example, necessary to surround the join point execution code in a *try-catch* block which will roll back the transaction when an exception is thrown. Fourth, the *method* attribute is set to *call*. The main difference between *call* and *execute* is that the join point context is different. This has an impact on the use of the *withincode* pointcut, for example, and on the information available in the reflective join point variable. The *withincode* pointcut is only used in combination with *call* in NFComp. This allows the precise selection of the exact execution context, i.e., the respective method calling the join point method. For *execution* only *cflowbelow* is used, which may lead to unintended matches in the control flow, e.g., when the targeted method is executed again on a higher level of the method call stack. In addition to the other attributes, a list of the key-value pairs for middleware service configuration is required. The transaction service must know which method it should execute in the context of the transaction. Hence, *param0* is set to the *$thismethod* variable. Furthermore, it must know on which object this method is executed. Thus, *param1* is set to the *$thismethod.target* variable. Finally, also the parameters must be passed to the method which is achieved by setting *param2* to *$thismethod.args*.

For black and gray box mapping, NFComp relies on the Apache Synapse Enterprise Service Bus, and thus the generator must produce an appropriate configuration for the ESB,

which enforces the composition specified in the model. Therefore, a set of XML configuration files is generated, one for enabling the proxy in front of the purchase order web service and others for the sequences of mediators to be executed. In case of black box mapping, the proxy configuration shown in Listing 4.10 is generated. In this configuration, there is a *switch* mediator with *cases* for the different operations of the web service. For example, in the case of *getAvailableItemsPerCategory*, the mediator sequence *getAvailableItemsPerCategory_AuthDecryptVerifyLogCache* is executed, which in turn executes the respective mediators for authentication, signature verification, logging and caching. In addition to the main proxy configuration file, there is one sequence configuration file per association and one sequence configuration file per mapped action and activity.

```
<proxy xmlns="http :// ws.apache.org/ns/synapse" name="PurchaseOrderService" transports ="http , https ">
  <target >
    <inSequence>
      <switch source="get-property('OperationName')">
        <case regex=" getAvailableItemsPerCategory ">
          <sequence key="getAvailableItemsPerCategory_AuthDecryptVerifyLogCache"/>
        </case>
        <case regex="sendInvoice">
          <sequence key="sendInvoice_DecryptVerify"/>
        </case>
        <case regex="sendPurchaseOrder">
          <sequence key="sendPurchaseOrder_AuthDecryptVerifyLog"/>
        </case>
      </switch>
      <send>
        <endpoint>
          <address uri="http://localhost:7080/../PurchaseOrderService"/>
        </endpoint>
      </send>
    </inSequence>
  <outSequence>
      <switch source="get-property('OperationName')">
        <case regex=" getAvailableItemsPerCategory ">
          <sequence key="getAvailableItemsPerCategory_CacheSignEncrypt"/>
        </case>
        <case regex="sendInvoice">
          <sequence key="sendInvoice_SignEncrypt"/>
        </case>
        <case regex="sendPurchaseOrder">
          <sequence key="sendPurchaseOrder_SignEncrypt"/>
        </case>
      </switch>
      <send/>
    </outSequence>
    </ target >
  <publishWSDL key="PurchaseOrderService_wsdl"/>
</proxy>
```

Listing 4.10: Proxy Configuration for PurchaseOrderService

In gray box mapping, there is an additional proxy configuration per partner service of the purchase order process. Listing 4.11 shows the configuration generated for the *SchedulingService* with its two operations *sendShippingSchedule* and *requestProductionScheduling*. As can be seen, there is a *switch* mediator for evaluating the *processIdMeta* property in the intercepted message. If this property has the value *PurchaseOrder*, i.e., the purchase order process was the sender of this message, the next *switch* mediator will be executed. This nested *switch* mediator checks the name of the activity which caused the message to be sent. If this is the *sendShippingSchedule* task for example, the *sendShippingSchedule_Log* sequence is executed.

```
<proxy xmlns="http :// ws.apache.org/ns/synapse" name="SchedulingService">
  <target>
    <inSequence>
      <collectctx />
      <switch source="get-property('processIdMeta')">
        <case regex="PurchaseOrder">
          <switch source="get-property(' activityIdMeta ')">
            <case regex="sendShippingSchedule">
              <sequence key="sendShippingSchedule_Log"/>
            </case>
            <case regex=" requestProductionScheduling ">
              <sequence key="requestProdSched_sendMessageInOrder"/>
            </case>
          </switch>
        </case>
      </switch>
      <send>
        <endpoint>
          <address uri="http :// localhost :8480/ axis2 / services / DiscountService . DiscountService /"/>
        </endpoint>
      </send>
    </inSequence>
    <outSequence>
      <switch source="get-property('processIdMeta')">
        <case regex="PurchaseOrder">
          <switch source="get-property(' activityIdMeta ')">
            <case regex="sendShippingSchedule">
              <sequence key="sendShippingSchedule_Log"/>
            </case>
          </switch>
        </case>
      </switch>
      <send/>
    </outSequence>
  </target>
  <publishWSDL key="SchedulingService_wsdl"/>
</proxy>
```

Listing 4.11: Proxy Configuration for SchedulingService

In the gray box proxy configuration, the *collectctx* mediator is responsible for extracting meta data such as *processIdMeta* and *activityIdMeta* from the message. The purchase order BPMN2 process has previously been transformed into a WS-BPEL process to be executed on the

Apache ODE BPEL server. This WS-BPEL process is automatically instrumented by additional copy activities (among others) in order to add the metadata to outgoing messages required by the proxy.

In white box view, the model is transformed into AspectJ code. Listing 4.12 shows the generated code for the transaction actions (monitoring code omitted). The first pointcut *ws_execution()* selects the top level operation of the purchase order web service *sendPurchaseOrder* and *getAvailableItemsByCategory*. This pointcut is used to instantiate the aspect once per invocation of a top level operation. This allows the storage of instance variables such as the identifier of the transaction context or the context object itself in the middleware service object instances which are assigned to the *txservice* and *monitorservice* attributes. A list of pointcuts follows which is generated out of the non-functional associations in the mapping diagram. Each pointcut selects a set of join points; for example, *participateInTx_executeUpdate* selects the call of all methods with the name *executeUpdate* defined in the *javax.persistence.Query* independently of its signature. Furthermore, *withincode* selects only join points from within the *createNewCategory* method and the *cflowbelow* keyword selects only those join points which are called in the control flow of *getAvailableItemsByCategory*. This respects the modeling semantics defined in the mapping diagram. For each non-functional action, an advice is generated which is of the type that has been chosen for the respective non-functional association, e.g., *around*, for the *participateInTx* action. The advice for this action invokes the *participate* method of the transaction service passing the three parameters which have been configured in the middleware mapping phase.

```
1  public aspect NFA percflow(ws_execution()) {
2
3    WSMonitoring monitorservice = new WSMonitoring();
4    WSTransaction txservice  = new WSTransaction();
5
6    public pointcut ws_execution():
7      execution(* PurchaseOrderPT.sendPurchaseOrder(..) )  ||
8      execution(* PurchaseOrderPT.getAvailableItemsByCategory (..) );
9
10   public pointcut  startTransaction_createNewCategory ():
11     execution(* ItemsStore.createNewCategory(..) )
12     && cflowbelow(execution(* PurchaseOrderPT.getAvailableItemsByCategory(..) ))
13     && !cflowbelow(execution(* WSTransaction.*(..) ));
14
15   public pointcut endTransaction_createNewCategory():
16     execution(* ItemsStore.createNewCategory(..) )
17     && cflowbelow(execution(* PurchaseOrderPT.getAvailableItemsByCategory(..) ))
18     && !cflowbelow(execution(* WSTransaction.*(..) ));
19
20   public pointcut  participateInTx_executeUpdate ():
21      call(* javax. persistence .Query.executeUpdate (..) )
22     && withincode(* purchaseorder. ItemsStore .createNewCategory (..) )
23     && cflowbelow(execution(* PurchaseOrderPT.getAvailableItemsByCategory(..) ))
24     && !cflowbelow(execution(* WSTransaction.*(..) ));
25
26   before():  startTransaction_createNewCategory (){
```

```
27      try{
28        txservice.begin("/Header/CoordinationContext/Identifier", "java:comp/UserTransaction",
29        "/Header/CoordinationContext/Expires");
30      }catch(Exception e) {...}
31    }
32
33    after(): endTransaction_createNewCategory(){
34      try{
35        txservice.end();
36      }catch(Exception e) {...}
37    }
38
39    Object around(): participateInTx_executeUpdate(){
40      try{
41        return txservice.participate(((MethodSignature)(thisJoinPoint.getSignature())).getMethod(),
              thisJoinPoint.getTarget(), thisJoinPoint.getArgs());
42      }catch(Exception e) {...}
43    }
44  }
```

Listing 4.12: Generated AspectJ Code for Transactions

The implementation of the *participate* method is shown in Listing 4.13. It first checks whether the transaction object has been initialized (which is performed by the *begin* method not shown in the listing) and if so, it invokes the join point method using Java's reflection mechanism. The invocation of the join point method is surrounded by a *try-catch* block in order to roll the transaction back, when an error occurs.

```
1  public class WSTransaction {
2
3    private UserTransaction transaction;
4
5    public Object participate(Method method, Object target, Object args[]) throws Exception{
6
7      if (transaction == null)
8        throw new Exception("Transaction not started");
9
10     try{
11       return method.invoke(target, args);
12     }
13     catch(Exception e){
14       setTransactionState(TransactionState.MUST_ROLLBACK);
15       throw new Exception(e);
16     }
17   }
18 }
```

Listing 4.13: Implementation of Method Participate

Ortiz and Hernandez In the approach of Ortiz and Hernandez [68], the platform independent model used for the specification of extra-functional properties is turned into a platform-specific

model. There are three different types of PSMs, though, depending on the nature of the respective property. When there is the need to develop additional functionality in order to enforce the property, the aspect-oriented metamodel should be used. When the extra-functional property needs to be described declaratively, the policy-based metamodel should be used. Finally, the SOAP-tag-based metamodel should be used whenever a new SOAP tag should be added to the message header in order to enforce the extra-functional property. There is a model-to-model transformation from the PIM into the respective PSM. After the transformation, missing information can be complemented manually. Then, the model-to-text transformation can be initiated. The aspect-oriented metamodel is transformed into AspectJ code, the policy-based metamodel is transformed into policies and policy attachments and the SOAP-tag-based model is transformed into SOAP handlers adding the SOAP header to the message. There is a repository for the "well-known" extra-functional properties to generate complete and executable AspectJ aspects. For other extra-functional properties, only a skeleton is generated which must be complemented manually. An example for an aspect generated for the authenticate property is shown in Listing 4.14.

```
1  public aspect sendPurchaseOrder_authenticate {
2
3    declare precedence: sendPurchaseOrder_authenticate, sendPurchaseOrder_decrypt, sendPurchaseOrder_verify,
         sendPurchaseOrder_log;
4
5    pointcut sendPurchaseOrder_authenticate_P(int param_1):execution(execution (public PurchaseOrderService.
         sendPurchaseOrder)(Invoice) && args(customerInfo, purchaseOrder));
6
7    Invoice around(CustomerInfo customerInfo, PurchaseOrder purchaseOrder): sendPurchaseOrder_authenticate_P(
         customerInfo, purchaseOrder){
8      Invoice result;
9      try{
10       if(ServerHandler_sendPurchaseOrder_authenticate.get("operationName").compareTo("sendPurchaseOrder") == 0
11         && ServerHandler_sendPurchaseOrder_authenticate.get("propertyName").compareTo("authenticate") == 0
12       ){
13         <Functionality to be completed>
14       }
15       else
16         result = proceed(customerInfo, purchaseOrder);
17     }catch(Exception e){
18       ...
19     }
20     return result;
21   }
22 }
```

Listing 4.14: Aspect Generated from the Aspect-Oriented Metamodel by Ortiz and Hernandez [68]

In the approach of Ortiz and Hernandez, there is no concept of dedicated middleware services. The implementation logic is directly generated into the AspectJ aspects or must be added manually. In the policy- or SOAP-tag-based metamodel PSM, the logic is implemented by SOAP handlers which act as message interceptors. In the case of policy, the handler is configured by the policy definition. In the case of the SOAP-tag-based metamodel, the logic is directly generated into the handler itself.

Sec-MoSC In Sec-MoSC [88], the annotated BPMN model is transformed into a WS-Policy configuration based on XML. There is support for 10 security-related *NF-Actions* so far. Depending on the action, standard WS-Security assertions or custom Sec-MoSC assertions are used. The standard WS-Security assertions are interpreted by the Apache Rampart[9] module for Apache Axis. The custom assertions are processed by a self-designed Sec-MoSC module implementing access control and logging. Listing 4.15 shows such a policy for encryption, signing and access control. Sec-MoSC provides the implementation logic for the *NF-Actions* inside the Apache Axis handlers, which are configured via the generated policy documents.

```
<wsp:Policy wsu:Id="POPolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wssp:SignedElements>
        <wssp:XPath>/S:Envelope/S:Body</wssp:XPath>
    </wssp:SignedElements>
    <wssp:EncryptedElements>
       <wsssp:XPath>/S:Envelope/S:Body</wssp:XPath>
      </wssp:EncryptedElements>
      <secmosc:SecMoscConfig>
        <secmosc:restrictAccess>
          <secmosc:restrictPartner policyType="allow">
            <secmosc:destinationEndpoint
              address="http://localhost/PurchaseOrderService" />
          </secmosc:restrictPartner>
        </secmosc:restrictAccess>
      </secmosc:SecMoscConfig>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 4.15: WS-Policy Generated from the Sec-MoSC Model [88]

AO4BPEL In AO4BPEL [14], for each requirement defined in the deployment descriptor, an aspect is generated. For particular requirements (the composite ones such as *InOrder* for reliable messaging or atomic transactions), a set of aspects is generated. These aspects define pointcut and advice elements. The pointcut is derived from the selector's XPath expression. For atomic actions (requirements for which only one aspect needs to be generated), the pointcut expression is equal to the selector's XPath. In case of composite actions, the XPath expression is extended depending on the type of aspect being generated. Listing 4.16 shows the generated aspect for starting a new reliable sequence with *inOrder* semantics. Listing 4.17 shows the generated aspect for sending messages as part of this sequence. The *assign* activity copies additional parameters to the input message of the reliable messaging service. For the reliable messaging aspects shown, these parameters are hard-wired in the aspect generation templates and make use of the reflective variables of AO4BPEL. These variables allow the access of data from the current join point such as the *scopeid* of the surrounding *scope* activity and so on. However,

[9] http://axis.apache.org/axis2/java/rampart/

for other aspects, for example in case of logging, parts of these assign activities are replaced, depending on the generator input. If, for example, someone has configured the log level in the deployment descriptor, this selected level is generated into the *from* part of the copy expression. The action to middleware service mapping is hard-wired into the generator templates. Whenever a new middleware service is to be introduced, a new aspect generator template must be written. If an existing middleware service needs to be replaced by a new one, the respective template needs to be changed accordingly.

The middleware services in AO4BPEL are all realized as web services. There is a security, reliable messaging, transaction and logging web service. Web services realizing composite actions (for example the transaction and the reliable messaging service) are stateful. They store the context information about the reliable sequence or the current transaction locally in the respective service. Hence, these web service operations take an identifier as an additional parameter in order to assure that it is executed in the right context, i.e., if the message to be sent with *inOrder* assurance belongs to the associated reliable sequence. The strategy to generate the identifiers is to use the id of the structured join point activity (in Listing 4.16 for example the *scopeid*).

```
1  <aspect name="s0_semantics_inOrder_start">
2    ...
3    <pointcut contextCollection ="true" name="s0inOrder">/process[@name="PurchaseOrder"]/sequence[@name="
         mainsequence"]</pointcut>
4    <advice type="before">
5      <bpws:sequence>
6        <bpws:assign>
7          <bpws:copy>
8            <bpws:from part="enclosedIsInonly" variable ="ThisJPActivityExt"/>
9            <bpws:to part="inonly" variable ="inputMessage"/>
10         </bpws:copy>
11         <bpws:copy>
12           <bpws:from part="enclosedPartnerEndpoints" variable ="ThisJPActivityExt"/>
13           <bpws:to part="endpoints" variable ="inputMessage"/>
14         </bpws:copy>
15         <bpws:copy>
16           <bpws:from part="scopeid" variable ="ThisJPActivity"/>
17           <bpws:to part="seqId" variable ="inputMessage"/>
18         </bpws:copy>
19       </bpws:assign>
20       <bpws:invoke inputVariable ="inputMessage"
21                    name="ReliableMessagingService_startNewSequenceWithExactlyOnceHelper"
22                    operation ="startNewSequenceWithExactlyOnceHelper" outputVariable="outputMessage"
23                    partner ="rmService" portType="rms:ReliableMessagingService"/>
24     </bpws:sequence>
25   </advice>
26 </aspect>
```

Listing 4.16: Aspect for Creating a Multi-Party Reliable Sequence for Purchase Order

```
<aspect name="s0_semantics_inOrder_add">
  ...
  <pointcut contextCollection ="true" name="compoundActivity">
    /process/sequence[@name="packageSequence"]/invoke | /process[@name="PurchaseOrder"]/sequence[@name="
        mainsequence"]/reply
  </pointcut>
  <advice type="around soapmessageout">
    <bpws:sequence>
      <bpws:assign>
        <bpws:copy>
          <bpws:from part="message" variable="soapmessage"/>
          <bpws:to part="message" variable="inputMessage"/>
        </bpws:copy>
        <bpws:copy>
          <bpws:from part="scopeid" variable=" ThisJPActivity "/>
          <bpws:to part="seqId" variable="inputMessage"/>
        </bpws:copy>
      </bpws:assign>
      <bpws:invoke inputVariable="inputMessage" name="ReliableMessagingService_addToReliableSequence"
          operation="addToReliableSequence" outputVariable="outputMessage"
          partner="rmService" portType="rms:ReliableMessagingService"/>
      <bpws:assign>
        <bpws:copy>
          <bpws:from part="addToReliableSequenceReturn" variable="outputMessage"/>
          <bpws:to part="newmessage" variable="newsoapmessage"/>
        </bpws:copy>
      </bpws:assign>
    </bpws:sequence>
  </advice>
</aspect>
```

Listing 4.17: Aspect for Adding a Message to a Reliable Sequence for Purchase Order

4.4.5.2 Feature Comparison

Criteria There are several features suitable for comparison of the relevant approaches in this phase. Firstly, there are different execution languages for composite web services, for example, process execution languages such as WS-BPEL or BPMN2 and general purpose programming languages such as Java or C#. The use of a dedicated process execution language is rated with 1 feature point.

Non-functional actions can be implemented as web services, Java classes or aspects and can be configured using WS-Policy, for example. The action realization, however, is not rated because there are benefits and drawbacks with all of the mentioned strategies. The integration of non-functional actions is also achieved differently. Handler approaches use an interception mechanism and are tightly integrated with the SOAP framework used. The advantage is that this allows for the integration of new actions at runtime (rated with 1 feature point). Alternatively, byte code enhancement/modification is used by many aspect-oriented approaches. However, this usually integrates actions at design time (0 feature points). A proxy, similar to handlers, also intercepts the invocation of web services; however, the proxy component can also be installed remotely. This allows the separation of web service implementation and integration code. Consequently, different programming languages could be used for proxy and web service. The type of integration mechanism is not rated in terms of feature points.

It is not only web services that should be enhanced by additional non-functional concerns in order to be enterprise-ready; the clients also need to be equipped with similar facilities. For example, in the case of security, the client must decrypt the encrypted response message of a web service. The challenge is to support the web service consumer to create client code which uses NFA mappings compatible to the service's NFA mapping.

In the approaches investigated in this subsection, there are two general strategies for middleware mapping, i.e., the configuration of NFAs. The first approach is to access configuration data from within the NFA implementation. This can be done by accessing reflective variables such as *thisJoinPoint* in AspectJ or *ThisJPActivity* in AO4BPEL from within the advice code. The second approach is to map the data which is required for the implementation of an NFA to its interface, i.e., the data is injected into the NFA implementation via parameters. In the first approach, all data is available to the NFA, whereas in the second approach, only explicitly mapped data is accessible.

In general, there are two types of data an NFA must access: constant values or variables pointing to a context-dependent value (for example the intercepted message). It is not obvious which approach is better (and hence it is not rated in terms of feature points); however, the second approach is more suitable when the NFA is loosely coupled or even running remotely from the functional code. In this case it makes no sense to transport all available data to the component implementing the NFA. Moreover, the implementation of the NFA does not depend on the technology which has been used to integrate it with the functional code (e.g., aspects).

Finally, an approach for the composition of non-functional concerns should allow a flexible and easy integration of new, unforeseen NFAs. This can be achieved through the support for integrating external middleware services by configuration. That is, the modeler can configure the use of arbitrary new NFA implementations as long as they conform to a certain type of interface, e.g., WSDL. If this is not supported by configuration, a user who wants to integrate a new type of NFA must put great effort into the investigation of the underlying framework in order to extend it; for instance, she must extend or modify the code generator.

NFComp In NFComp, composite web services are modeled in BPMN2 and are then transformed into executable WS-BPEL code. Actions are either realized as web services or as local mediators being part of the Enterprise Service Bus. In the white box view, also plain Java classes can be used. Middleware services are integrated via a proxy which stands in front of the target web service or process, intercepts the messages and invokes the middleware services according to the specification model. In the case of white box services, AspectJ is used for the integration of the Java classes implementing the non-functional actions. AspectJ uses a byte code weaving approach to integrate advice code into join points. Also clients can make use of the proxy at the consumer side. With help of the *inverse* interdependency, even the action composition logic can automatically be determined for the client. If middleware services are realized as web services, they can easily be integrated into the proxy. In the middleware mapping specification,

the WSDL of the middleware web service can be imported and associated with a non-functional action. The parameters to be mapped to the middleware service are flexible, because arbitrary key-value-pairs can be chosen. The key should match the web service parameter name. The value can either be a constant value or a context variable which the proxy will replace at runtime. No new generation code must be written in case of new middleware web services being introduced (although this must be done in the case of new local mediators).

Ortiz and Hernandez In the approach of Ortiz and Hernandez [68], no process execution language is used. Only the black box view is supported, and thus WSDL is the only visible input to the specification model. Actions are realized as AspectJ aspects or with WS-Policy. They are integrated into the web service using handlers or byte code weaving depending on the type of PSM being used. Not only the provider side, but also the client side can be modeled. In this case, client stubs are generated with handlers integrating the code for realizing the extra-functional property. In the aspect-based generation approach the *thisJoinPoint* variable can be used to access context data. Other data such as configuration constants can be defined as additional properties in the stereotype, e.g., to configure the path to the log file. This additional data available in the model can be processed by the code generator. However, for each new NFA implementation or property the generator must be extended. New middleware services cannot be integrated flexibly because they are directly part of the advice code or implemented inside the handler. If a new extra-functional property is added to the repository of the well-known properties, the transformation templates need to be extended to generate the code. Otherwise only a skeleton can be generated.

Sec-MoSC In Sec-MoSC [88], composite web services are modeled by BPMN2 but are transformed into WS-BPEL later. Actions are turned into WS-Policy documents which have been extended to support additional *NF-Attributes* such as access control. The integration is realized with handlers inside the Apache ODE BPEL Engine. There is no client-side support. Middleware services are realized as a combination of policy documents and handlers being configured via these documents. Configuration parameters are mapped via key-value-pairs, but support only constant values. The mapped configuration is then transformed into (custom) policy assertions. The mapping logic is hidden inside the code generator, hence, for each new type of configuration property, the generator must be extended. The modules which enforce the policies have access to context data via the Apache Axis *MessageContext* class. In order to change the handler or the generated policy to integrate whole new middleware services, the transformation logic must also be adapted.

AO4BPEL AO4BPEL [14] uses WS-BPEL directly as the executable process language for composite web services. Middleware Services are solely realized as web services. The reason is that an AO4BPEL advice is written in WS-BPEL code. In WS-BPEL, web services provide the functionality and can easily be integrated. The weaving of the advice code into the WS-BPEL

engine is directly implemented in the engine itself. This is why AO4BPEL can only be used with WS-BPEL engines which have been extended with this special weaving mechanism (for example Apache ODE for AO4BPEL2[10]). The weaving is done at runtime, which allows the deployment of new aspects when the process is already running. There is no dedicated support for web service clients. In AO4BPEL, context data can be accessed via reflective variables such as *ThisJPActivity*. However, it is also possible, through the generator, to add constant configuration values to the generated aspects. The configuration can be added to the requirement elements in the deployment descriptor; however, the code for the mapping is implemented in the generator. For the integration of new middleware services, new aspects must be written. For supporting the generation of these aspects from the deployment descriptor, the aspect generator must be extended.

Summary Table 4.12 outlines the features discussed and feature points. Features which have not been rated use an italic font type. NFComp received the most feature points in this category because it supports all rated features. It is the only approach that allows adding new middleware services purely by configuration and supports not only web service providers but also web service consumers.

Feature/Criterion	**NFComp**	**Ortiz & Hernandez**	**Sec-MoSC**	**AO4BPEL**
Process Exec Language	BPEL	–	BPEL	BPEL
Action Realization	WS/Mediator/ Java	Aspect/ Policy	Policy	WS
Action Integration	Proxy/ ByteCode	ByteCode/ Handler	Handler	Engine Code
Weaving Type	Runtime	Design time	Design time	Runtime
Client side support	Yes	Yes	No	No
Context data	ParamMapping/ Consts and Vars	CtxVar	ParamMapping/ Constants	CtxVar
Config new mw services	Yes	No	No	No
Feature Points	4	1	1	2

Table 4.12: Feature Comparison in the Code Generation Phase

4.4.6 Summary

In summary, the case study and feature comparison showed that the purchase order scenario can be implemented with different approaches. However, the approaches perform differently with respect to a variety of features. Those features have systematically been analyzed, and their support has been rated using feature points. Table 4.13 outlines the feature points for the analyzed works per phase and aggregates them to achieve a total comparison. The results show

[10] `http://www.stg.tu-darmstadt.de/research/ao4bpel`

that NFComp has received by far the most feature points in total, although SCA, Sec-MoSC and AO4BPEL also received quite a good rating.

Phase	NFComp	NFR	Sec-MoSC	SCA	ProcNFL	AO4BPEL	Ortiz & Hernandez
Req. Spec.	6.5	7.0	5.5	**7.5**	5.0	4.5	0.0
Action Def.	**6.0**	0.0	3.0	3.0	0.0	3.5	1.5
Action Com.	**8.0**	0.0	1.5	1.0	0.0	1.5	2.0
Action Map.	**7.0**	0.0	2.0	2.5	0.0	1.0	1.0
Code Gen.	**4.0**	0.0	1.0	0.0	0.0	2.0	1.0
All Phases	**31.5**	7.0	13.0	14.0	5.0	12.5	5.5

Table 4.13: Overall Feature Points Comparison

4.5 Limitations

This section outlines the limitations of the NFComp approach in terms of general limitations and limitations with respect to the current implementation.

4.5.1 General Limitations

When composing non-functional actions with each other, two types of compositions play an important role: the vertical and the horizontal one. On one hand, the vertical composition defines the execution order of non-functional actions when being applied to the same functional point. This is covered by the non-functional activity concept in NFComp. On the other hand, the horizontal composition defines processes of non-functional actions similar to functional processes. Horizontal composition is implemented by the composite action concept. However, the vertical composition is emphasized in NFComp, at least during the mapping phase. Non-functional activities are directly mapped to functional targets whereas composite actions are hidden (there is no graphical representation in the mapping model). In contrast, composite non-functional actions are used to validate the mapping. The reason for the **unbalanced representation of vertical versus horizontal composition** is the simplification of the graphical model in favor of a better readability and thus also better understandability. Representing all three dimensions functional process, vertical and horizontal composition at once would require the use of a three-dimensional model or the mapping of two aspects to the same dimension. This kind of model would be hard to comprehend.

In NFComp, there is **no separation between the selection of a set of non-functional actions and the ordering of these actions**. Each is represented by a non-functional activity which must be mapped to its functional target in order to be activated. Thus, there is no possibility to define a general global ordering of non-functional actions. This shortcoming may lead to redundancy. However, in most cases redundancy can be avoided by using nested non-

functional activities, which is allowed in NFComp. Thus, in this way general global orderings can be encapsulated by a non-functional activity, which then can be reused in different other non-functional activities. Nevertheless, this strategy is limited, because it is not possible to insert additional non-functional activities inside such a general global ordering definition. This is, for example, necessary in order to be able to adapt a general ordering for a particular functional execution point. Furthermore, it is not assured that a global ordering definition is actually enforced. Such an activity must be used explicitly inside another activity in order to be activated. The decision against direct support for general orderings in NFComp has been taken because — as already motivated in Section 2.3 — the ordering may depend on a concrete use case or be specific to a particular functional subject. Hence, if general ordering definitions were to be supported, this would also imply the requirement to be able to override/extend such general definitions with additional/adapted actions. A set of strategies for workflow inheritance has been outlined by van Aalst and Basten [91]. Workflow inheritance is a non-trivial research topic, and in most cases the use of nested activities turns out to be sufficient. This is why NFComp does not support general global ordering definitions.

The white box approach, in its current state, is not as mature as the black and gray box approach. The reverse engineered behavioral model is obtained by a static program analysis [103]. However, the produced BPMN2 process is **not yet an exact representation of the Java code**. Firstly, parameters are currently not used for an exact match for the method invocation in the case of multiple methods with the same name defined for the same class. Hence, the name of a task should be *<classname>. <methodname>(<parametertypes>)* to improve the precision. However, there are particular restrictions with programming languages supporting dynamic dispatch, such as Java. Firstly, it is not clear what the type of the runtime object passed as the parameter actually is. The reason is that there could be overloaded methods with the same name and the same number of parameters and the actual method to be invoked could be chosen at runtime. This is only critical in programming languages with multiple dispatch (Java supports single dispatch only). However, secondly, programming languages with single dispatch decide at runtime on which object a method should be called. Hence, with polymorphism, the content of a variable could be of different subtypes of the variable types. Thus, the invoked method is not known at compile time and cannot be determined with static program analysis. Another weakness of the white box approach is that parallelism is not yet supported and is complex to implement.

4.5.2 Limitations of the Current Implementation

NFComp's architecture for black and gray box view relies on the concept of remote proxies. The proxy component has the same interface as the target service and is responsible for message interception and non-functional action composition. The advantage of the proxy used as a remote component is that it allows for the separation of service implementation and action implementation, composition and integration. This enables developers to use their programming language

of choice to implement non-functional actions. Furthermore, already developed middleware services can be combined with all kinds of web services. The common handler approach does not support this, because there is a strong coupling between web service implementation and middleware service implementation. However, despite the high flexibility, the remote proxy approach introduces some drawbacks. The first drawback is the **limited applicability**. The service provider must install the proxy in her service hosting environment; because, if the proxy implements middleware concerns such as security, a secure channel between proxy and web service should be used. The installation in the hosting environment is not always possible due to policy restrictions by the provider's company. The second drawback is the **performance reduction**. All messages sent to a web service must pass the proxy component (potential bottleneck) which requires (de-)serializing the message additionally. Nonetheless, this bottleneck can be compensated for clustering of the proxy component. In order to determine the concrete performance reduction, performance tests have been conducted. In the test setup, Apache Axis2 on Apache Tomcat 6 has been used to host web services, and Apache Synapse 1.2 standalone server has been used as enterprise service bus deploying the proxies. The test results can be found in Table 4.14. In the test scenario the client sends a message with a size of 439 bytes to a web service that replies with a 1450 bytes response. The results shown in the table are the average response time for 1000 invocations. The first column shows the response time for a simple web service invocation (WS), the second column shows the invocation with an additional Synapse proxy (WS + Proxy) and column three presents the results for a web service invocation with proxy and Log mediator (WS + Proxy + Log) that logs the full request message. Finally, the fourth column shows the response time for the same web service with logging integrated directly in the web service implementation (WS + Log).

Abs/Rel	**WS**	**WS + Proxy**	**WS + Proxy + Log**	**WS + Log**
Absolute	11.86ms	14.013ms	15.4ms	12.176ms
Relative	100%	118%	130%	103%

Table 4.14: Performance Reduction Through the Use of Proxies

The results show that there is a slight overhead of 18%, when using the proxy in front of the target web service. The main reason is that the XML message needs to be deserialized and serialized by the proxy in addition to the web service. The *Log* mediator requires additional time to log the whole XML SOAP message to the console. Hilsdale and Hugunin [43] found out that the best AspectJ implementation of logging adds a 22% overhead relative to the hand-coded logging implementation. NFComp is comparable to these results because it introduces 26% overhead (WS + Proxy + Log divided by WS + Log).

NFComp has **limited support for loops**. There are three concepts in the approach where loops play a role: non-functional activities, composite non-functional actions and composite web services. In non-functional activities, loops are not supported at all. There is no need to execute a particular non-functional action several times, at least none of the investigated use

cases required this. Composite non-functional actions and composite web services are more likely to use loops. However, loops are challenging for the validation mechanism. NFcomp uses an iteration-based validation procedure enumerating all possible execution paths. This number could explode if unstructured loops were used. Furthermore, the Prolog implementation presumes that for each opening gateway there is exactly one corresponding closing gateway. This restriction may be violated by unstructured loops. This is why NFComp only supports structured loops in composite non-functional actions and composite web services. The same limitation also applies to WS-BPEL processes in general; cf. [2].

In the implementation of NFComp's white box view approach, there are some limitations with the use of AspectJ. Its **pointcut language is not powerful and precise enough** to select exactly those join points that are targeted by non-functional associations in the NFComp mapping diagram (also referred to as history-based advice or trace matching [1]). To achieve a precise join point selection it is not enough to select only the method call based on the target method signature, but also the call stack (i.e., from which method it is called), and the exact position in the code must be considered. Although the first requirement can be addressed with *withincode* and *cflow* pointcuts, the second requirement is not enforceable by means of AspectJ. As long as there is only one method call to a particular method, there is no problem at all; however, it is not possible to select one out of two calls to the same method from within the same method. This limitation exists because AspectJ, on one hand, has access to reflective join point information such as the method arguments, the method name and signature and so on; but on the other hand has limited access to the call context and has no information about methods or statements executed before or after the join point. Hence, one cannot restrict a pointcut to an exact position in a method such as "*match only if another method has been called before or after the join point*". Consequently, the mapping diagram could express requirements which cannot be supported by AspectJ. However, most of the web service implementations, investigated for the white box view, did not contain such ambiguous method calls. Nevertheless, the modeler should at least be aware of these situations. This can, for example, be achieved by an automatic check in the mapping editor which, in the case of such a problem, would warn the modeler.

4.6 Conclusion

In this chapter, the NFComp approach has been evaluated with respect to the requirements which led to the invention of this approach, high-level criteria to assess engineering approaches in general and supported features compared to other approaches. The requirements evaluation revealed that all specification and realization requirements are supported by NFComp except S6.2: Data Flow Specification and negligible limitations in E3: Quantification and E5: Transparent Weaving with distributed WS. NFComp also supports and fosters all of the identified evaluation criteria. It has a wide scope of application, low specification complexity, high standard compliance (especially with respect to BPMN2), flexible extensibility and strong validation

mechanisms to ensure specification correctness. Furthermore, it is more expressive in terms of workflow patterns than related approaches; fosters separation of concerns and supports multiple users, roles, views and execution environments. The case study and feature comparison showed how to apply NFComp to the purchase order scenario and how other approaches would implement it. In addition, a set of features has been identified in order to compare NFComp to other approaches. This feature comparison showed that NFComp received more than 31 feature points compared to the second top-rated approach with 14 feature points. However, there are nonetheless a few limitations with NFComp such as limited support for loops, unbalanced mapping of horizontal and vertical composition mappings and slightly reduced applicability and performance introduced by the proxy component. The performance reduction is, however, comparable to other aspect-oriented approaches.

In summary, this chapter showed that NFComp is a holistic approach, performing very well with respect to the applied evaluation techniques. There are only a few weaknesses which are negligible compared to the benefits gained by applying NFComp for NFC composition in web services.

Chapter 5

Summary and Future Work

5.1 Summary

The main goal of this thesis was to address unresolved shortcomings in non-functional concern composition for web services and other component-based applications by inventing a holistic and model-driven approach focusing on specification as well as enforcement issues and taking different views on web services into account.

The motivation for research in this topic mainly originated in the research project PREMIUM|Services, in which a marketplace offered a set of middleware services to be composed with pricing web services. This set of middleware services must be composed in a flexible manner, but no industry standard had been available at this time to address this problem in an appropriate way. Thus, in this thesis, a systematic study of state of the art in academia and industry has been conducted. The research projects and the analyzed related work lead to a set of requirements which have been categorized by specification and realization/enforcement issues. Revisiting related works and comparing them to the requirements revealed that most of the analyzed approaches either focus on specification or realization issues and do not support the requirements sufficiently. The severest weaknesses identified were missing support for fine-grained composition of NFAs, dynamic control flow between NFAs, interdependencies between NFAs and platform independence.

New concepts as well as an implementation for a novel approach called NFComp has thus been developed in this thesis. The decision to make this approach model-driven was carried out to cope with the inherent complexity of NFCs and web services. Further motivation has been the wide and successful adoption of process-orientation in the context of web services, making the behavior or workflow of services a first-class entity in the form of models, for example with BPMN2 or WS-BPEL. The innovative idea in this thesis was to allow a modeler not only the specification of workflow logic for business services but also to describe the behavior of non-functional concerns in a similar way.

To accomplish this, NFComp delivers two contributions: an abstract and generic framework for component-based applications and a concrete, instantiable one for web services. The abstract framework can be used to develop new applications of NFComp for other component-based applications besides web services. The concrete web-service-specific part can be applied to web services from different views: black, gray and white box. NFComp is a highly structured approach with distinct phases where particular models are created by different roles. Each view extends the abstract framework by its specific features, providing a holistic and modular solution.

The decision of the view taken for a web service depends on its nature and the non-functional requirements. In black box view, atomic web services and non-functional concerns can be addressed, whereas gray and white box also support composite non-functional concerns. The gray box view can be used for composite web services defined in BPMN2. The white box view is suitable when it is necessary to look inside a web service but no composition language has been used or if a component has been written in Java which is not necessarily a web service.

After the specification of a model, NFComp helps to turn this model into executable code. The implementation details depend on the concrete view. The black and gray box view implementation makes use of a proxy which intercepts messages to the web service and invokes the middleware services in the order specified in the model. The gray box approach, however, is more complex and requires a slight instrumentation of the WS-BPEL code which implements the composite web service. In the white box view, a completely different strategy is chosen, because also internal classes can be enhanced with middleware functionality. For Java-based web services, AspectJ aspects are generated to enforce the modeled non-functional behavior.

An essential feature for modeling non-functional action compositions is the validation of the model at design time. For this purpose, knowledge on the nature of non-functional actions with respect to their composability can be stored in the model. An action may have a relationship called interdependency of different type which allows for the definition of dependencies and control flow constraints. The constraints imposed by interdependencies can be used to validate non-functional activities as well as the mapping with respect to composite non-functional actions.

However, an important insight which has been gained in this thesis is that there are also cross-concern interdependencies. These interdependencies are hard to determine, because different non-functional domain experts must collaborate in this case. To mitigate this difficulty, the data impact of actions has been analyzed systematically, finding that data impact can be used to infer new interdependencies automatically. This facilitates the enrichment of the interdependency model and enables the identification of more cross-concern interdependencies, which strongly improves the precision of the validation mechanism. In addition to the classical validation approach, a guided modeling procedure has been introduced. This procedure allows the creation of conflict-free compositions by suggesting only the next valid modeling steps during the modeling of non-functional tasks.

To illustrate the applicability and to compare the NFComp approach directly to others, a case study has been conducted. This case study demonstrated how NFComp and other approaches can be used to implement the purchase order scenario for different views. This showed that different views and some of the phases are also only supported by parts of the related work, proving that NFComp is more holistic than other approaches. In the context of the case study, a detailed feature comparison showed that NFComp is the most powerful approach in this area. To make the comparison more tangible, features have been rated with feature points. NFComp received 31 feature points, which is quite an impressive result compared to the second best rated work with 14 points. The case study with the feature comparison included was only one out of three evaluation techniques used to assess the quality of NFComp. Another technique, an analysis of the originating requirements which lead to the approach, showed that most of the requirements have been successfully addressed. Furthermore, an evaluation against high-level quality criteria has been conducted. In this evaluation, NFComp performed very well, especially in terms of separation of concerns.

The NFComp approach achieved good evaluation results; however, there are also a few drawbacks with respect to concepts and their implementation. Most of the drawbacks are trade-offs in favor of obtaining certain properties such as performance versus platform independence or vertical versus horizontal composition. In the next section, an outline is presented on ways in which particular drawbacks may be addressed in the future.

5.2 Future Work

There are two areas of future work discussed in this section. Firstly, an analysis is made as to how the current limitations of NFComp could be addressed in the future, and, secondly, a discussion is conducted on widening the scope of this approach from web services to web applications.

5.2.1 Addressing Limitations

The limitation *unbalanced representation of vertical versus horizontal composition* could be addressed by introducing an additional view on the model. This view could have a horizontal focus instead of a vertical one. This would allow the user to switch between the different views depending on her requirements. Another idea would be to have a combined model showing the effective process with functional and non-functional actions included. This model could be derived from the BPMN definition of the composite service plus mapping and action model. The resulting process would reflect the overall execution order. On one hand, such a model does not strictly separate functional from non-functional concerns, but, on the other hand, it could improve understandability. Furthermore, it it should be read-only because it would be completely derived from other models.

In the current implementation of NFComp there is a limitation with loops. However, there are several approaches (for example, Mendling et al. [58]) which try to mitigate this limitation by turning unstructured elements into structured ones. If such an approach could be applied in an automated way (even it would produce duplicated tasks), the BPMN processes containing unstructured loops could be transformed into processes with only structured loops before NFComp's validation mechanism is applied.

There are limitations with the behavioral model generation and the implementation of the white box view. The limitations in the behavioral model generation are the lack of parallelism and the ambiguity of method invocations in programming languages with dynamic dispatch (for example Java). The latter could be solved as follows: For each set of methods which are candidates for dynamic dispatch, a set S of subtasks is generated. The subtask representing a single method invocation is then replaced by an exclusive gateway pair defining n branches, each containing exactly one subtask from S. The limitation on parallelism is hard to address. For each *Thread* a new AND gateway must be created, and all method invocations in its *run()* method must be added to the branch. However, also objects of type *Runnable* can be executed concurrently. For synchronization there are multiple concepts such as the *synchronized* modifier and *wait* and *notify* methods on each object.

The problem with AspectJ's insufficient pointcut language could be improved by using the approach of Allan et al. [1]. They propose to extend AspectJ with the concept of *tracematches* being capable of selecting traces of events in a program. A *tracematch* defines one or more symbols (called events of interest), a pattern which consists of symbols and a piece of code being executed when the trace (more specifically the pattern) matches. A symbol comprises an identifier, an advice type such as before or after, and a pointcut. For each BPMN pool representing a process of method invocations, a *tracematch* could be produced in the following way: For each task a new symbol is created with a pointcut matching the respective method call. Then, for each task associated with a non-functional action, a pattern is created describing the exact sequence of methods that must be invoked before the method of the associated task is invoked. This would allow for distinguishing the same types of method calls even across different process branches.

5.2.2 Widening Scope

The focus of the research project InDiNet (which was the second research project, besides PREMIUM|Services, this thesis contributed to) is on cloud computing, more specifically platform as a service (PaaS). In this context, the platform is constituted by a set of services called platform services. These platform services can be used by a cloud consumer in order to run her software applications in the cloud. A web service is an example for such an application; however, it is also common to run web applications with a graphical user interface on such a platform.

The main difference between services and classical web applications is that the latter is directly consumed by humans whereas the former is consumed by machines. Hence, a web ap-

plication as such poses new requirements with respect to non-functional concerns. Firstly, there is the graphical user interface which is consumed through browsers using the HTTP protocol. Secondly, the content delivered via this transport protocol is not SOAP, but may be HTML, Flash, Javascript, all kinds of XML, or even a mixture of all of these. Thirdly, there is no interface definition and the functional behavior is distributed among different web sites. The black box and gray box view are not suitable with those constraints given. However, the white box view could be applied instead. If, for example, a web application was written in Java, it could be reverse-engineered to a BPMN process, non-functional actions could be defined and AspectJ aspects could be generated out of this.

Another interesting research direction would be the support for crosscutting concerns which are not non-functional concerns. This idea originates in the nature of platform services which adhere more to crosscutting concerns than to non-functional ones. Examples are billing services, image conversion services (offered by Google AppEngine[1] for example) or also database and persistence services (provided by most cloud providers). These services, however, do no always play a role during a particular service invocation but could define their own independent lifecycle.

For example, billing is a functionality which crosscuts many applications but is executed on a monthly basis and not based on a particular event (e.g., the user pushed a particular button or sent a certain form) in the application. A billing service would make use of an accounting service which would collect all events a customer must be charged for. The billing service would filter the events and generate an invoice. However, for this scenario, there must be an interface between accounting and billing service which can be used for the data transfer. This interface is different from the one which is used for the integration of the platform service into the target application and could be called platform-to-platform (p2p) service interface.

[1] https://developers.google.com/appengine/

Appendix A

Curriculum Vitae

- *2009 - 2013*
  Technische Universität Darmstadt (Darmstadt University of Technology)
  Ph.D. Student in the Software Technology Group of Prof. Mira Mezini
  Research Associate at SAP Research, Darmstadt
  Graduated with a doctoral degree in June 2013
- *2006 - 2009*
  Software engineer and consultant at UBL Informationssysteme GmbH, Neu-Isenburg
- *2000 - 2006*
  Technische Universität Darmstadt (Darmstadt University of Technology)
  Studies in Computer Science. Graduated as Diplom-Informatiker
  (comparable to an MSCS or master's degree in computer science/information technology)
- *1997 - 2000*
  Gymnasialzweig, Friedrich-Ebert-Schule, Kooperative Gesamtschule, Pfungstadt
  (Sixth form / secondary school, Friedrich-Ebert Comprehensive School in Pfungstadt, Germany)
  Graduated with Abitur, final secondary-school examination
  (degree requirement for university study in Germany)
- *1991 - 1997*
  First through fifth forms, secondary school at Friedrich-Ebert Comprehensive School, Pfungstadt, Germany
- *1987 - 1991*
  Erich-Kästner-Schule, Pfungstadt (primary school in Pfungstadt, Germany)
- *April 18, 1981*
  Born in Wiesbaden, Germany

Bibliography

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," *SIGPLAN Not.*, vol. 40, no. 10, pp. 345–364, Oct. 2005.

[2] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. Kevin, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu, "Web Services Business Process Execution Language Version 2.0," April 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[3] F. Baligand and V. Monfort, "A concrete solution for web services adaptability using policies and aspects," in *Proceedings of the 2nd international conference on Service oriented computing. ICSOC '04.* New York, NY, USA: ACM, 2004, pp. 134–142.

[4] S. Battle, "Boxes: Black, White, Grey and Glass Box Views of Web-Services," Digital Media Systems Laboratory, HP Laboratories Bristol, Tech. Rep., 02 2003. [Online]. Available: http://www.hpl.hp.com/techreports/2003/HPL-2003-30.pdf

[5] M. Beisiegel, N. Kavantzas, A. Malhotra, G. Pavlik, and C. Sharp, "SCA Policy Association Framework," in *Service-Oriented Computing–ICSOC 2006*, ser. Lecture Notes in Computer Science, A. Dan and W. Lamersdorf, Eds., vol. 4294. Springer, 2006, pp. 613–623.

[6] L. Bergmans and M. Aksit, "Composing software from multiple concerns: A model and composition anomalies," in *Proceedings of the 22nd International Conference on Software Engineering. ICSE 2000.*, 2000, p. 8, Workshop on Multi-Dimensional Separation of Concerns in Software Engineering.

[7] ——, "Composing crosscutting concerns using composition filters," *Communications of the ACM*, vol. 44, no. 10, pp. 51–57, 2001.

[8] J. Bonér, "AspectWerkz-dynamic AOP for Java," in *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[9] D. Box and F. Curbera, "Web Services Addressing (WS-Addressing)," August 2004. [Online]. Available: http://www.w3.org/Submission/ws-addressing/

[10] M. Braem, K. Verlaenen, N. Joncheere, W. Vanderperren, R. V. D. Straeten, E. Truyen, W. Joosen, and V. Jonckers, "Isolating Process-Level Concerns Using Padus," in *Business*

Process Management, ser. Lecture Notes in Computer Science, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102. Springer, 2006, pp. 113–128.

[11] A. Charfi, R. Khalaf, and N. Mukhi, "QoS-Aware Web Service Compositions Using Non-intrusive Policy Attachment to BPEL," *Service-Oriented Computing–ICSOC 2007*, pp. 582–593, 2007.

[12] A. Charfi and M. Mezini, "AO4BPEL: An Aspect-oriented Extension to BPEL," in *World Wide Web*, 2007, pp. 309–344.

[13] A. Charfi, H. Müller, and M. Mezini, "Aspect-Oriented Business Process Modeling with AO4BPMN," in *Proceedings of the Sixth European Conference on Modelling Foundations and Applications. ECMFA 2010.*, ser. Lecture Notes in Computer Science, vol. 6138. Springer, Jun. 2010, pp. 48–61.

[14] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini, "Reliable, Secure, and Transacted Web Service Compositions with AO4BPEL," in *Proceedings of the 4th European Conference on Web Services. ECOWS'06.* IEEE Computer Society, Dec. 2006, pp. 23–34.

[15] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 1, pp. 9–36, 1976.

[16] S. Chollet and P. Lalanda, "An Extensible Abstract Service Orchestration Framework," in *Proceedings of the 2009 IEEE International Conference on Web Services. ICWS '09.* Washington, DC, USA: IEEE Computer Society, Jul. 2009, pp. 831–838.

[17] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," W3C, March 2001.

[18] L. Chung and J. do Prado Leite, "On non-functional requirements in software engineering," in *Conceptual Modeling: Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, Eds. Springer Berlin / Heidelberg, 2009, vol. 5600, pp. 363–379.

[19] J. Clark and S. DeRose, "XML Path Language (XPath) Version 1.0," November 1999. [Online]. Available: http://www.w3.org/TR/xpath/

[20] W. Clocksin and C. Mellish, *Programming in PROLOG*. Springer, 2003.

[21] C. Courbis and A. Finkelstein, "Towards aspect weaving applications," in *Proceedings of the 27th International Conference on Software Engineering. ICSE 2005.*, May 2005, pp. 69–77.

[22] D. Coward and Y. Yoshida, "Java Servlet Specification, Version 2.4," Sun Microsystems, Tech. Rep., 2003. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr154/

[23] F. Curbera, M. J. Duftler, R. Khalaf, W. A. Nagy, N. Mukhi, and S. Weerawarana, "Colombo: Lightweight middleware for service-oriented computing," *IBM Systems Journal*, vol. 44, no. 4, pp. 799–820, 2005.

[24] L. M. Cysneiros and J. C. S. do Prado Leite, "Using UML to reflect non-functional requirements," in *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '01. IBM Press, 2001, pp. 2 (1–15).

[25] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: an environment for goal-driven requirements engineering," in *Proceedings of the 19th international conference on Software engineering. ICSE '97.* New York, NY, USA: ACM, 1997, pp. 612–613.

[26] D. Davis, A. Karmarkar, G. Pilz, S. Winkler, and Ümit Yalçinalp, "Web Services Reliable Messaging Protocol (WS-ReliableMessaging) Version 1.2," February 2009. [Online]. Available: http://docs.oasis-open.org/ws-rx/wsrm/v1.2/wsrm.html

[27] D. Davis, A. Karmarkar, G. Pilz, and Ümit Yalçinalp, "Web Services Reliable Messaging Policy Assertion (WS-RM Policy) Version 1.2," February 2009. [Online]. Available: http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.2-spec-os.html

[28] P. Durr, L. Bergmans, and M. Aksit, "Reasoning about Semantic Conflicts between Aspects," in *Proceedings of the 2nd European Interactive Workshop on Aspects in Software. EIWAS 05.*, Brussels, Belgium, September 2006, pp. 10–18.

[29] ——, "Static and Dynamic Detection of Behavioral Conflicts Between Aspects," in *Seventh International Workshop on Runtime Verification*, Vancouver, Canada, March 2007, pp. 38–50.

[30] Eric S. K. Yu, "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering," in *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering. RE '97.* Washington, DC, USA: IEEE Computer Society, 1997, p. 226.

[31] Erradi and Maheshwari, "AdaptiveBPEL: A policy-driven middleware for flexible Web services composition," in *Proceedings of the EDOC Middleware for Web Services Workshop. MWS 2005.*, 2005.

[32] R. Filman and D. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Workshop on Advanced Separation of Concerns in conjunction with OOPSLA*, October 2000, pp. 21–35.

[33] J. Fox and J. Jürjens, "A Framework for Analyzing Composition of Security Aspects," in *Proceedings of Methods for Modelling Software Systems. MMOSS 2006.*, ser. Dagstuhl Seminar Proceedings, vol. 06351, Aug. 2006.

[34] X. Franch, "Systematic formulation of non-functional characteristics of software," in *Proceedings of the Third International Conference on Requirements Engineering*, 1998, pp. 174 –181.

[35] M. Galster and E. Bucherer, "A taxonomy for identifying and specifying non-functional requirements in service-oriented development," in *Proceedings of the 2008 IEEE Congress on Services - Part I. SERVICES '08.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 345–352.

[36] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[37] K. Ganesan, S. K. Mohalik, and C. Raj, "A distributed aspect model for composite service," in *Proceedings of the HiPC 2008 International Workshop on Service-Oriented Engineering and Optimization*, Dec. 2008.

[38] M. Glinz, "Rethinking the Notion of Non-Functional Requirements," in *Proceedings of the Third World Congress for Software Quality. 3WCSQ'05.*, 2005, pp. 55–64.

[39] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The SmartFrog configuration management framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 16–25, Jan. 2009.

[40] O. M. Group, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Version 1.1," April 2008. [Online]. Available: http://www.omg.org/spec/QFTP/1.1/PDF/

[41] S. Heinzl and B. Schmeling, "Using XML schema to improve writing, validation, and structure of WS-policies," in *Proceedings of the 2010 ACM Symposium on Applied Computing. SAC 2010.* ACM, Mar. 2010, pp. 2422–2429.

[42] M. Henkel, G. Bostroem, and J. Wäyrynen, "Moving from Internal to External Services Using Aspects," *Interoperability of Enterprise Software and Applications*, pp. 301–310, 2006.

[43] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," in *Proceedings of the 3rd international conference on Aspect-oriented software development. AOSD '04.* New York, NY, USA: ACM, 2004, pp. 26–35.

[44] M. Hmida, R. Tomaz, and V. Monfort, "Applying AOP concepts to increase Web services flexibility," in *Proceedings of the International Conference on Next Generation Web Services Practices. NWeSP 2005.*, Aug. 2005, pp. 169–174.

[45] W. L. Hürsch and C. V. Lopes, "Separation of concerns," Northeastern University, Boston, USA, Tech. Rep., 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5223

[46] IBM Research, "Web Intermediaries (WBI)," 2011. [Online]. Available: http://www.almaden.ibm.com/cs/wbi/wbi-poster.pdf

[47] M. Indulska, M. zur Muehlen, and J. Recker, "Measuring method complexity: The case of the business process modeling notation," *BPMcenter.org*, 2009. [Online]. Available: http://bpmcenter.org/wp-content/uploads/reports/2009/BPM-09-03.pdf

[48] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

[49] S. Kallel, A. Charfi, T. Dinkelaker, M. Mezini, and M. Jmaiel, "Specifying and monitoring temporal properties in web services compositions," in *Proceedings of the 2009 Seventh IEEE European Conference on Web Services. ECOWS '09.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 148–157.

[50] S. Katz, "Aspect Categories and Classes of Temporal Properties," Feb. 2006, vol. 3880, pp. 106–134.

[51] S. Katz and M. Sihman, "Aspect Validation Using Model Checking," in *Verification: Theory and Practice*. Springer, 2004, pp. 391–392.

[52] G. Kiczales, J. Lamping, A. Mendhekar *et al.*, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming. ECOOP 97*, ser. LNCS, vol. 1241. Springer, 1997, pp. 220–242.

[53] P. Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley, 2003.

[54] R. Laddad, *AspectJ in Action*. Manning Publications, 2003.

[55] K. Lawrence, C. Kaler, A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist, "WS-SecurityPolicy 1.3," OASIS Standard, February 2009. [Online]. Available: http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.doc

[56] M. Little and A. Wilkinson, "Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1," November 2004. [Online]. Available: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec/wstx-wsat-1.1-spec.html

[57] S. Mellor, M. Balcer, and I. Foreword By-Jacoboson, *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[58] J. Mendling, K. Lassen, U. Zdun *et al.*, "Transformation strategies between block-oriented and graph-oriented process modelling languages," in *Multikonferenz Wirtschaftsinformatik (MKWI 2006)*, vol. 2. Vienna, Austria: GITO-Verlag Berlin, 2006, pp. 297–312.

[59] N. C. Mendonça, C. F. Silva, I. G. Maia, M. A. F. Rodrigues, and M. T. de Oliveira Valente, "A Loosely Coupled Aspect Language for SOA Applications," *International Journal of Software Engineering and Knowledge Engineering*, vol. 18, no. 2, pp. 243–262, 2008.

[60] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using non-functional requirements: A process-oriented approach," *IEEE Transactions on Software Engineering*, vol. 18, pp. 483–497, 1992.

[61] I. Nagy, L. Bergmans, and M. Aksit, "Composing Aspects at Shared Join Points," in *NetObjectDays (NODe/GSEM)*, ser. LNI, R. Hirschfeld, R. Kowalczyk, A. Polze, and M. Weske, Eds., vol. 69. Erfurt, Germany: GI, September 2005, pp. 19–38.

[62] M. Naveed, M. K. Abdullah, K. Rashid, and H. F. Ahmad, "Representing Shared Join Points with State Charts: A High Level Design Approach," in *Transactions on Engineering, Computing and Technology*, 2006.

[63] H. F. Nielsen, N. Mendelsohn, J. J. Moreau, M. Gudgin, and M. Hadley, "SOAP version 1.2 part 1: Messaging framework," W3C, W3C Recommendation, Jun. 2003.

[64] M. Nishizawa, S. Chiba, and M. Tatsubori, "Remote pointcut: a language construct for distributed AOP," in *Proceedings of the 3rd international conference on Aspect-oriented software development. AOSD '04.* New York, NY, USA: ACM, 2004, pp. 7–15.

[65] Object Management Group, "Meta Object Facility (MOF) Core Specification Version 2.0," Tech. Rep., January 2006. [Online]. Available: http://www.omg.org/spec/MOF/2.0/PDF/

[66] ——, "OMG Unified Modeling Language (OMG UML), Superstructure, V2.3," Tech. Rep., May 2010. [Online]. Available: http://www.omg.org/spec/UML/2.3/Superstructure/PDF

[67] ——, "Business Process Model and Notation (BPMN) 2.0," January 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/PDF

[68] G. Ortiz and J. Hernandez, "A Case Study on Integrating Extra-Functional Properties in Web Service Model-Driven Development," in *Proceedings of the Second International Conference on Internet and Web Applications and Services. ICIW '07.* Washington, DC, USA: IEEE Computer Society, May 2007, p. 35.

[69] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," *International Conference on Web Information Systems Engineering*, pp. 3 (1–31), 2003.

[70] M. Parastoo, "An approach for empirical evaluation of model-driven engineering in multiple dimensions," in *6th European Conference of Modelling Foundations and Applications. ECMFA 2010*, Paris, France, June 2010, pp. 6–17.

[71] E. Pulvermueller, A. Speck, J. Coplien, M. D'Hondt, and W. DeMeuter, "Feature interaction in composed systems," in *Proceedings of the Workshops on Object-Oriented Technology. ECOOP '01*. Budapest, Hungary: Springer, June 2001, pp. 86–97.

[72] A. Rashid and A. Moreira, "Domain models are not aspect free," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Springer Berlin / Heidelberg, 2006, vol. 4199, pp. 155–169.

[73] N. Rosa, P. Cunha, and G. Justo, "ProcessNFL: A Language for Describing Non-functional Properties," in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences. HICSS'02*, vol. 9. Washington, DC, USA: IEEE Computer Society, January 2002, pp. 282.2 (1–10).

[74] M. Rossi and S. Brinkkemper, "Complexity metrics for systems development methods and techniques," *Information Systems*, vol. 21, no. 2, pp. 209–227, 1996.

[75] M. Sánchez and J. Villalobos, "A flexible architecture to build workflows using aspect-oriented concepts," in *Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling. AOM '08.* New York, NY, USA: ACM, 2008, pp. 25–30.

[76] F. Sanen, E. Truyen, and W. Joosen, "Managing concern interactions in middleware," in *Distributed Applications and Interoperable Systems*, June 2007.

[77] B. Schmeling, A. Charfi, S. Heinzl, and M. Mezini, "A survey on non-functional concerns in web services," *International Journal of Web Information Systems (IJWIS)*, vol. 8, no. 1, pp. 5–31, 2012.

[78] B. Schmeling, A. Charfi, M. Martin, and M. Mezini, "Towards conflict-free Composition of non-orthogonal, non-functional Behavior," in *24th International Conference on Advanced Information Systems Engineering (CAiSE'12)*. Gdansk, Poland: Springer, June 2012.

[79] B. Schmeling, A. Charfi, and M. Mezini, "Non-functional Concerns in Web Services: Requirements and State of the Art Analysis," in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, Paris, France, November 2010.

[80] ——, "Composing Non-Functional Concerns in Composite Web Services," in *IEEE International Conference on Web Services (ICWS'11)*. Washington DC, USA: IEEE Computer Society, July 2011.

[81] B. Schmeling, A. Charfi, R. Thome, and M. Mezini, "Composing Non-Functional Concerns in Web Services," in *The 9th European Conference on Web Services (ECOWS'11)*. Lugano, Switzerland: IEEE Computer Society, September 2011.

[82] P. Shaker and D. K. Peters, "Design-level detection of interactions in aspect-oriented systems," in *Proceedings of the 20th European Conference on Object-Oriented Programming*, July 2006.

[83] B. Shannon, "Java Platform, Enterprise Edition (Java EE) Specification, v5," Sun Microsystems, Tech. Rep., 2006. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr244/

[84] K. Siau and M. Rossi, "Evaluation techniques for systems analysis and design modelling methods–a review and comparative analysis," *Information Systems Journal*, vol. 21, no. 3, pp. 249–268, 2011.

[85] S. Singh, J. Grundy, J. Hosking, and J. Sun, "An Architecture for Developing Aspect-Oriented Web Services," in *Proceedings of the Third European Conference on Web Services. ECOWS '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 72–82.

[86] E. Soeiro, I. S. Brito, and A. Moreira, "An XML-Based Language for Specification and Composition of Aspectual Concerns," in *Proceedings of the International Conference on Enterprise Information Systems. ICEIS 2006.*, Y. Manolopoulos, J. Filipe, P. Constantopoulos, and J. Cordeiro, Eds., 2006, pp. 410–419.

[87] I. Sommerville, *Software Engineering (7th Edition)*. Addison-Wesley Longman, 2004.

[88] A. Souza, B. Silva, F. Lins, J. Damasceno, N. Rosa, P. Maciel, R. Medeiros, B. Stephenson, H. Motahari-Nezhad, J. Li, and C. Northfleet, "Incorporating security requirements into service composition: From modelling to execution," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, L. Baresi, C.-H. Chi, and J. Suzuki, Eds. Springer Berlin / Heidelberg, 2009, vol. 5900, pp. 373–388.

[89] S. Supakkul and L. Chung, "A UML Profile for Goal-Oriented and Use Case-Driven Representation of NFRs and FRs," in *Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications. SERA '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 112–121.

[90] A. M. Turing, "On Computable Numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, pp. 230–265, 1936.

[91] W. van der Aalst and T. Basten, "Inheritance of workflows: an approach to tackling problems related to change," *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 125–203, 2002.

[92] W. Van Der Aalst and A. Ter Hofstede, "YAWL: yet another workflow language," *Information Systems*, vol. 30, no. 4, pp. 245–275, 2005.

[93] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering. RE '01.* Washington, DC, USA: IEEE Computer Society, Aug. 2001, pp. 249–262.

[94] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers, "Adaptive programming in JAsCo," in *Proceedings of the 4th international conference on Aspect-oriented software development. AOSD '05.* New York, NY, USA: ACM, Mar. 2005, pp. 75–86.

[95] A. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and Ü. Yalçinalp, "Web Services Policy 1.5 – Framework," W3C, Tech. Rep., 2007. [Online]. Available: http://www.w3.org/TR/ws-policy/

[96] ——, "WS-PolicyAttachment V1.5," Sept. 2007. [Online]. Available: http://www.w3.org/TR/ws-policy-attach/

[97] B. Verheecke, W. Vanderperren, and V. Jonckers, "Unraveling Crosscutting Concerns in Web Services Middleware," vol. 23. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 42–50.

[98] I. Weber, A. Barros, N. May, J. Hoffmann, and T. Kaczmarek, "Composing Services for Third-party Service Delivery," in *Proceedings of the 2009 IEEE International Conference on Web Services. ICWS '09.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 823–830.

[99] P. Wohed, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and N. Russell, "Pattern-based Analysis of BPMN," December 2005.

[100] E. Wohlstadter, S. Tai, T. Mikalsen, J. Diament, and I. Rouvellou, "A Service-oriented Middleware for Runtime Web Services Interoperability," in *Proceedings of the IEEE International Conference on Web Services. ICWS '06.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 393–400.

[101] R. Yin, "The abridged version of case study research: Design and method." in *Handbook of Applied Social Research Methods*, D. J. R. Leonard Bickman, Ed. Sage Publications, Inc, 1998.

[102] U. Zdun, "Tailorable language for behavioral composition and configuration of software components," *Comput. Lang. Syst. Struct.*, vol. 32, pp. 56–82, April 2006.

[103] A. Zeller *et al.*, "Program analysis: A hierarchy," in *Proceedings of the ICSE Workshop on Dynamic Analysis. WODA 2003.*, 2003, pp. 6–9.

[104] M. Zenger, "Programming language abstractions for extensible software components," *Department of Computer Science, Lausanne*, March 2004, PhD thesis.